



# Turbo Pascal 6.0

## 编程技术与实例



北京希望电脑公司



363235

TP314  
0267

TP314  
0267

CS 52- Turbo Pascal 6.0  
编程技术与实例

刘明 张恒 编

北京希望电脑公司

## 内容简介

本书系统地介绍了 Turbo Pascal 的最新版本 Turbo Pascal 6.0 的语言特性和程序设计技术和技巧。全书分为十四章,内容包括 Turbo Pascal 6.0 的基本概念、数据类型、控制结构以及其它语言特性,包括集成开发环境、工具箱、视频显示和软、硬中断高级编程,面向对象编程以及 Turbo Pascal 最重要的增强—面向对象应用工具 Turbo vision 的应用实例。

本书内容丰富、完整,具有大量程序示例,深入浅出,易于掌握。既是高等院校学生的极好的计算机语言教学参考书,又是计算机系统和应用开发人员极好的指南手册。

5015557

## 前 言

N. Wirth 教授设计的 Pascal 语言是公认的定义严格,结构严谨,并且支持结构化程序设计方法学的一种高级语言,目前它是计算机专业的首选计算机教学语言,而且几乎国内外所有的大、中、小、微型计算机上都有它的编译器。

Turbo Pascal 是最流行的 Pascal 系统,是美国 Borland 公司推出的,可以在 PC DOS、MSDOS、CP/M80、CP/M86、OS/2 等操作系统支持的 IBM PC/XT、AT、286、386 PS/2 及其兼容机上运行,许多评论家认为 Turbo Pascal 已是事实上的微机 pascal 标准。

Turbo Pascal 有许多优良的特性,它设计精巧、用户界面友善,编译速度的运行效率高。它完全遵从 N. wirth 的标准 Pascal,也遵从 ISO 7185-1983 国际标准,而且还进行了大量的扩充,如利用计算机的硬件特性,存取绝对地址变量、内嵌汇编(机器)代码,结构常量等等,还提供了大量视频控制能力,如图形、颜色、窗口及声音等等。

另外, Turbo Pascal 具有一个集编辑、编译、连接、运行,求助,调试于一身的集成软件开发环境,支持重叠窗口、鼠标、菜单、对话框等交互手段。

Turbo Pascal 还支持可以分别编译的单元,有利于大型软件的开发。而且从 5.5 版本起,扩充了面向对象程序设计的设施,在 Turbo Pascal 6.0 中又有进一步改进和增强,尤其还提供了面向对象的应用工具 Turbo Vision,使人们可以充分表达模块化、信息隐藏、抽象代码共享等软件工程思想。

本书系统地介绍了 Turbo Pascal 的最新版本 6.0 的各种语言特点和优良特性,并通过大量程序实例阐明 Turbo Pascal 的高级编程技术和技巧,是一本难得的全面、丰富的 Turbo Pascal 6.0 程序设计教程和编程指南。

本书分为十四章,分别给出了基本概念、数据类型、语句和控制结构、集成开发环境,内存管理、文件、过程和库、工具箱、视频显示控制、软、硬中断处理、程序调试、面向对象编程等内容,最后一章还包括一个面向对象应用工具 Turbo Vision 的应用实例。本书内容详实,结构完整,深入浅出,易于掌握。

本书在编写整理的过程中,得到了多方而的帮助与支持,尤其要感谢李京同志为提高成书质量所作的大量工作。

本书的编写者虽然都是长期从事程序设计工作,并对操作系统,编译结构以及各类高级语言设施的特性有过一些研究,但由于时间短促,难免在选材和叙述上存在各种不足,欢迎读者指评指正。

# 目 录

<b>第一章 Turbo Pascal 编程的基本概念</b> .....	(1)
§ 1.1 Turbo Pascal 程序的一般形式 .....	(1)
§ 1.2 Turbo Pascal 与标准 Pascal .....	(2)
§ 1.3 程序结构 .....	(3)
§ 1.3.1 程序头与编译指令 .....	(3)
§ 1.3.2 数据部分 .....	(7)
§ 1.3.3 代码部分 .....	(9)
§ 1.3.4 包含文件 .....	(10)
§ 1.3.5 覆盖块 .....	(10)
§ 1.3.6 过程与函数 .....	(11)
<b>第二章 数据类型与表达式</b> .....	(14)
§ 2.1 常量 标准数据类型 .....	(14)
§ 2.2 常量 .....	(15)
§ 2.3 用户定义的数据类型 .....	(15)
§ 2.4 集合 .....	(17)
§ 2.5 数组 .....	(18)
§ 2.6 记录 .....	(19)
§ 2.7 turbo Pascal 中的表达式 .....	(25)
§ 2.7.1 算术运算 .....	(25)
§ 2.7.2 整型运算 .....	(27)
§ 2.7.3 算术函数 .....	(27)
§ 2.7.4 逻辑运算 .....	(28)
§ 2.7.5 集合运算 .....	(30)
§ 2.8 类型间的关系 .....	(31)
<b>第三章 程序的控制结构</b> .....	(33)
§ 3.1 程序的选择结构 .....	(33)
§ 3.2 程序的循环结构 .....	(40)
§ 3.2.1 For—Do 循环 .....	(40)
§ 3.2.2 Repeat—Until 循环 .....	(40)
§ 3.2.3 While—Do 循环 .....	(41)
§ 3.3 非结构分枝 .....	(42)

<b>第四章 Turbo Pascal 的集成开发环境</b> .....	(45)
§ 4.1 File 菜单 .....	(45)
§ 4.2 Edit 菜单 .....	(46)
§ 4.3 Search 菜单 .....	(47)
§ 4.3.1 Find...CTRL-QE .....	(47)
§ 4.3.2 Replace...CTRL-QA .....	(48)
§ 4.3.3 Search again CTRL-L .....	(48)
§ 4.3.4 Go to line number... ..	(48)
§ 4.3.5 Find Procedure .....	(49)
§ 4.3.6 Find error... ALT...F8 .....	(49)
§ 4.4 Run 菜单 .....	(49)
§ 4.5 Compile 菜单 .....	(49)
§ 4.6 Debug 菜单 .....	(50)
§ 4.7 Options 菜单 .....	(51)
§ 4.7.1 Compiler... ..	(51)
§ 4.7.2 Memory Size... ..	(52)
§ 4.7.3 LinrRer... ..	(52)
§ 4.7.4 Directories... ..	(52)
§ 4.7.5 Environment .....	(52)
§ 4.7.6 Save Options... ..	(53)
§ 4.7.7 Retrieve Options... ..	(53)
§ 4.8 window 菜单 .....	(53)
<b>第五章 指针与动态内存分配</b> .....	(55)
§ 5.1 Turbo Pascal 的内存分配 .....	(55)
§ 5.2 堆和指针 .....	(57)
§ 5.3 链表 .....	(60)
§ 5.4 树 .....	(63)
§ 5.5 操作符 .....	(65)
<b>第六章 文件</b> .....	(66)
§ 6.1 文本文件 .....	(66)
§ 6.2 类型文件 .....	(69)
§ 6.3 无类型文件 .....	(72)
§ 6.4 缓冲区 .....	(74)
§ 6.5 文件的删除与改名 .....	(75)
<b>第七章 外部过程、过程与函数库</b> .....	(77)
§ 7.1 嵌入代码 .....	(77)
§ 7.2 外部过程 .....	(79)
§ 7.3 Turbo Debugger .....	(84)
§ 7.4 视频显示基本例程 .....	(86)

§ 7.5 带缓冲字符串输入 .....	(91)
§ 7.6 大字符串的处理 .....	(100)
§ 7.7 算术函数 .....	(103)
§ 7.8 文件加密 .....	(105)
<b>第八章 Turbo Pascal 工具箱 .....</b>	<b>(113)</b>
§ 8.1 数据库工具箱 .....	(113)
§ 8.2 图形工具箱 .....	(120)
§ 8.3 编辑工具箱 .....	(129)
§ 8.4 数值方法工具箱 .....	(138)
<b>第九章 编程技术 .....</b>	<b>(148)</b>
§ 9.1 字符串 .....	(148)
§ 9.2 递归 .....	(152)
§ 9.3 DOS 设备 .....	(160)
§ 9.4 合并 .....	(161)
§ 9.5 排序 .....	(164)
§ 9.6 搜索 .....	(168)
<b>第十章 视频: 文本显示与图形 .....</b>	<b>(170)</b>
§ 10.1 视频显示的基本概念 .....	(170)
§ 10.2 使用 Turbo Pascal 显示文本 .....	(174)
§ 10.2.1 方式、颜色和位置控制 .....	(174)
§ 10.2.2 直接存取视频存贮区 .....	(178)
§ 10.2.3 Turbo Pascal 的窗口程序设计 .....	(186)
§ 10.2.3.1 弹出窗口 .....	(186)
§ 10.2.3.2 多逻辑屏幕和弹出窗口 .....	(188)
§ 10.3 Turbo Pascal 的图形单元 (GRAPH) .....	(203)
§ 10.3.1 描点 .....	(203)
§ 10.3.2 画线 .....	(206)
§ 10.3. 圆、直线和图形模式的综合使用 .....	(209)
§ 10.3.4 图形文本 .....	(217)
§ 10.3.5 多边形及填彩 .....	(219)
<b>第十一章 DOS: 软中断与硬中断 .....</b>	<b>(223)</b>
§ 11.1 DOS 与 BIOS 服务 .....	(223)
§ 11.2 DOS 与中断 .....	(225)
§ 11.3 软中断: DOS 单元与操作系统公共服务 .....	(228)
§ 11.3.1 Turbo Pascal 的 DOS 单元 .....	(229)
§ 11.3.1 DOS 单元的常量与类型 .....	(229)
§ 11.3.1.2 DOS 单元的 DOSError 变量 .....	(232)
§ 11.3.1.3 DOS 单元的过程与函数 .....	(232)
§ 11.3.2 直接访问 BIOS 和 DOS 服务 .....	(235)

§ 11.3.2.1 磁盘驱动器服务 .....	(237)
§ 11.3.2.2 视频服务 .....	(247)
§ 11.3.2.3 时间和日期功能 .....	(252)
§ 11.3.2.4 报告换标状态 .....	(263)
§ 11.3.3 使用 DOS 单元中其它例程 .....	(265)
§ 11.4 硬中断, 远程通信与 TSR 实现 .....	(278)
§ 11.4.1 编写中断处理程序 .....	(278)
§ 11.4.2 PC 远程通信及程序 .....	(280)
§ 11.4.3 内存驻留程序(TSR)实现 .....	(293)
§ 11.4.3.1 解决再入问题 .....	(294)
§ 11.4.3.2 用 Turbo Pascal 实现 TSR .....	(294)
<b>第十二章 优化与调试</b> .....	(300)
§ 12.1 控制结构的优化 .....	(301)
§ 12.2 其它优化方法 .....	(306)
§ 12.3 编译指令 .....	(310)
§ 12.4 调用与参数 .....	(313)
§ 12.5 Turbo Pascal 调试器 .....	(315)
<b>第十三章 对象</b> .....	(320)
§ 13.1 对象的概念 .....	(320)
§ 13.2 继承 .....	(322)
§ 13.3 封装 .....	(324)
§ 13.4 静态方法和虚拟方法 .....	(328)
§ 13.5 对象类型的兼容性 .....	(332)
§ 13.6 对象的动态分配 .....	(332)
§ 13.7 多态性 .....	(334)
<b>第十四章 Turbo Vision</b> .....	(339)



# 第一章 Turbo Pascal 编程的基本概念

Pascal 程序设计语言是目前应用最广泛的计算机语言之一。它是由瑞士的沃斯(Niklaus Wirth)教授于一九七一年开发出来的。其命名是为了纪念波兰数学家 Pascal。开发 Pascal 语言的目的之一是为了培养程序员,使他们能够熟练地运用结构化程序设计的技能。

在传统 Pascal 语言的基础上, Borland 进一步把它发展为丰富多彩的语言—Turbo Pascal 语言。除具有传统 Pascal 的特点外, Turbo Pascal 还有各种各样的工具集。今天 Turbo Pascal 已经成为微机世界的标准 Pascal。

## § 1.1 Turbo Pascal 程序的一般形式

Pascal 是一种结构化的语言,每一部分都有特定的功能。它有严格的变量说明,程序结构和控制流的规则。结构化程序强调把程序分解为易于管理的小块,再把这些小块按照相互关系结合到一个程序。

GOTO 语句通常会破坏程序的结构化,这条语句允许程序员在程序中跳到任何地方。这样增加了程序的灵活性,但也使程序结构变得复杂,可读性差,不易调试。Pascal 语言提供了丰富的程序控制结构语句,尽量减少 GOTO 语句的使用。这样既增加了程序的可读性,也有助于减少非结构化程序流程带来的不可预测的错误结果。

Turbo Pascal 程序的一般形式为:

```
程序头
    程序名
    编译指令
    数据部分
        常量说明
        类型说明
        变量说明
        标号说明
    代码部分
        过程
        函数
        程序块
    程序体
```

Pascal 程序可以只包括其中的部分,数据说明部分不是必需的,但程序体是必不可少的。程序体以 Begin 开始,以 End 结束。程序从第一个 Begin 开始执行,到最后一个 End 语句结束,但遇到 Halt 语句除外,每个语句之后用分号与下一个语句分开,但 Begin 语句之后和 End 语句之前不用分号。下面是一个程序例子:

```
Program Prognam;
Uses CRT;
```

```

Var
    Number1,
    Number2,
    Number3;
Begin
    Clrscr;
    Write (' Enter a number;');
    ReadLn(number1);
    Write('enter another number;');
    ReadLn(number2);
    number3 := number1+number2;
    Writeln;
    Writeln('Number1+Number2=', number3:10:3);
End.

```

这个程序是从键盘分别输入两个数，然后输出这两个数的和。

## § 1.2 Turbo Pascal 与标准 Pascal

标准 Pascal 有许多优点，但不能满足目前商业应用的发展需要。它缺少有效的输入输出功能，没有串类型等，都不利于它的使用。Turbo Pascal 是在标准 Pascal 上更新了的语言，它既有标准 Pascal 的逻辑结构，又有一个扩展工具集。

Pascal 通常称为强类型语言。不同类型的变量不能混淆。在赋值语句中，左右两边的变量类型必须匹配。例如，变量  $i$  定义为整型变量。则下面的语句为非法的

```
 $i := 1.0 + 2;$ 
```

因为值 1.0 是实型数，不能用于整型数的赋值。这不符合强类型语言的宗旨。强类型规则有助于避免程序设计中的错误。Turbo Pascal 中的强类型要求不如标准 Pascal 严格。但仍需遵守下列规则：

- \* 实型数不能直接用于整型变量或字节的赋值语句，必须先用 trunc 和 Round 标准函数把它们转换为整型数。
- \* 只有在长度和类型都一致时，一个数组才可对另外一个数组赋值。
- \* 传递给过程或函数的变量类型必须与过程说明的类型一致。但对于串类型可以用 { \$V- } 编译指令取消。

一般来讲，强类型有很多好处。但有时候不使用强类型，问题更好处理。使用类型的强制转换可以解决这个问题。类型的强制转换就是把一种类型的变量暂时作为另外一种类型处理。例如，有时可能要求出字符的 ASCII 值，直接把一个字符的值赋给一个整型变量是不允许的。但用类型强制转换可以做到。如

```
 $i := \text{Integer}(c);$ 
```

其中  $i$  是整型变量， $C$  是字符型变量，这条语句把  $C$  的二进制值作为整型数赋给  $i$ 。指针型变量被解释为整型变量和一个串变量。类型强制转换技术对指针特别有用，这样它可以用于任何类型的数据。

## § 1.3 程序结构

Turbo Pascal 程序主要由三部分组成：程序头，数据部分和代码部分。各部分有其特定的功能。

### § 1.3.1 程序头与编译指令

Turbo Pascal 程序的前二行一般为程序名和编译指令。程序名也可以没有，但是为了归档方便最好使用程序名。程序名标识了程序的名字以及在程序中是否使用了输入输出。典型的程序头如下：

```
Program ProgName(Input, Output)
```

程序名后可以带一个参数表，但对于 Turbo Pascal 程序不是必需的。

Turbo Pascal 编译器可带许多选择项，指导编译器如何工作。这些选择项称作编译指令。编译指令在 Turbo Pascal 程序中起着重要作用。它们控制着各种错误检查和输入/输出控制。要充分发挥 Turbo Pascal 的能力，就必须懂得如何使用选择项。编译指令大致分为三类：开关指令，参数指令，条件指令。

#### 开关指令

开关指令打开或关闭 Turbo Pascal 的某项特殊功能。开关指令由单个字母标识，大写字母与小写字母相同。如 S 指令控制堆栈检查，R 指令设置范围检查等等。开关指令的格式为一个 \$ 号后跟一个编译指令及一个正号(允许)，或负号(禁止)，这些指令用两种注释定界符限定。这两种定界符是带 \* 号的圆括号或花括号。如：

```
(* $I- *)
```

```
{ $I- }
```

```
{ $ s +, v -, r +, a + }
```

等都是有效的编译指令。前两个语句是相同的，都是禁止输入

输出错误检查。第三个例子说明了四个编译指令：允许堆栈溢出检查 S+，禁止变量字符检查 V-，允许范围检查 r+，及按字对齐数据 a+。在这个例子中，只在第一个指令前有 \$ 号。

编译指令的设置有两种方式。最简单的方式是由 Options

Compiler 菜单设置。这样设置的指令是所有程序和程序块的缺省值，所有未在源代码行说明的编译指令，都使用 Options

compiler 设置的值。

在程序头上说明的开关指令是全局性的，它们影响整个程序的编译。它们必须在第一个 Uses，常量，标号，类型，过程，函数或 Begin 之前，否则为局部性的，局部指令可以出现在程序中的任何地方，它们只影响以后的程序。

#### {A} 数据对齐

当数据对齐指令有效时({ \$ A + })，Turbo Pascal 保证每个大于 1 字节长的变量和有类型常数都在偶地址开始。这样对齐后，8086 系列微处理机可以快速访问内存，但也增加了存储数据所需的内存量。因为数据在字边界对齐后，有些字节会成为“死”字节。若程序使用的内存量必须考虑时，可禁止这个指令{ \$ A - }。

### {B} 布尔变量求值

Turbo Pascal 支持两种类型的布尔变量求值：完全布尔变量求值和短路布尔变量求值。短路布尔变量求值时，只要能确定整个表达式的结果时，就不再继续运算。在很多情况下，这可以大大加快程序的运行。例如：

$(a < b) \text{ And } (b > c)$

表达式，使  $a$  大于  $b$  时，不论  $b$  是否大于  $c$ ，其结果都是假。这时短路布尔变量求值即结果。使用  $\{ \$B- \}$  进行短路布尔变量求值，使用  $\{ \$B+ \}$ ，进行完全布尔变量求值。

### {D} 调试信息

使用允许调试信息指令  $\{ \$D+ \}$ ，Turbo Pascal 将产生可执行指令在源文件中定位所需要的信息。使用这些信息可以逐行检查一个程序，或者在出现运行错误时确定错误的位置。这些调试信息加在 TPU 文件的后面，使文件增大，但不会影响可执行代码的速度。若使用禁止指令  $\{ \$D- \}$ ，则不能单步执行程序，但可稍稍缩短编译时间并节省一些磁盘空间。

### E 仿真

有的计算机带有 8087 数字协处理器，另外一些计算机则没有。如果一个程序需要 8087 的数字精度，并且这个程序有可能在不带 8087 协处理器的计算机上运行。当允许仿真  $\{ \$E+ \}$  时，Turbo Pascal 检查是否安装了 8087 芯片。如果有，程序将使用协处理器。如果没有，程序将用主处理器实现 8087 运算。自然如果没有 8087 协处理器，计算将用较长时间，但是程序可在任何一种机器上运行。除非需要特别精确的数学结果，最好使用禁止仿真指令  $\{ \$E- \}$ 。

### F 强制远程调用

Turbo Pascal 支持多个代码段，每个单元一个代码段，主程序另有一个代码段。在一个单元或程序中调用函数或过程为近程调用，不需要改变代码段。一个单元中的语句调用另外一个单元中的过程为远程调用。近程调用要做的工作较少。远程调用牵涉到不止一个代码段，做的工作较多，执行速度也慢。Turbo Pascal 可以判断应该用近程调用还是用远程调用。有时候在可以做近程调用的地方，需要强制一个过程为远程调用。必须强制使用远程调用的环境是非同寻常的。必要时，可使用允许强制远程调用指令  $\{ \$F+ \}$ ，使一个过程为远程调用。

### I 输入/输出检查

I/O 错误可能是最常见的 Turbo Pascal 错误类型，也是最危险的错误之一。它们使看似正常运行的程序产生不可预测的结果。 $\{ \$I+ \}$  语句就是用于检查程序中的 I/O 错误。使用这条指令后，出现 I/O 错误将产生运行错误并停止执行程序。 $\{ \$I+ \}$  指令不是处理 I/O 错误的最好方法。更有效的方法是使用  $\{ \$I- \}$ ，用户自己发现 I/O 错误。

### L 局部符号信息

局部符号信息是指局部于一个单元或过程的变量和常量信息，Turbo Pascal 一般不保存这些符号的信息，因而在调试对话中不能看到或者改变它们的值。 $\{ \$L+ \}$  语句使 Turbo Pascal 产生并保存关于所有局部变量的信息以供调试程序时使用。L 指令与调试信息指令 D 共同使用时，结果如下：当禁止调试信息时，L 编译指令不起作用，Turbo Pascal 不保存任何调试信息。当允许 D 指令，禁止 L 指令，即  $(\{ \$D+ \}, \{ \$L- \})$ ，Turbo Pascal 保存所有符号的调试信息。

### N 数值处理

Turbo Pascal 处理浮点运算有两种方式：正常方式和 8087 方式。正常方式为 6 字节数据，不使用 8087 协处理器。8087 方式提供另外四种浮点数据类型，使用 8087 协处理器。 $\{ \$N$

一) 语句选择正常方式, {\$N+} 选择 8087 方式。N 指令可以与 E 指令共同使用。当两个指令都有效时, 即使没有 8087 协处理器, 程序也可以用 8087 方式处理, 即使用 {\$N+, E+} 时, 若装了 8087 芯片, 程序就用它进行浮点操作, 否则使用仿真程序。仿真程序具有相同的精度, 但速度较慢。若禁止仿真指令 {\$N+, E-}, 程序只能在装有 8087 协处理器的计算机上运行。

#### O 覆盖代码产生

在一个覆盖文件中引用一个单元时, 必须使用 {\$O+} 指令。它告诉 Turbo Pascal 产生管理这个覆盖单元所需的代码。这时不是必须覆盖这个单元, 只是使覆盖成为可能。反之, 除非 O 指令有效, 否则不能覆盖这个单元。

#### R 范围检查

Turbo Pascal 的大部分数据类型都有限制。一字节数据的最大值为 255。5 个元素的数组不能有 6 个元素。定义为十个字符的字符串不能有十一个字符。若超出了限制会产生一个范围错误。当使用 {\$R+} 指令后, Turbo Pascal 产生代码检查所有数组下标和赋值是否出界。发现范围错误后产生一个运行错误, 停止执行程序。如果使用 {\$R-}, 所有的出界赋值和下标都不报告, 其结果可能是灾难性的。

在一个好的程序中不应出现范围错误, 但在程序开发的早期却是很难避免的。所以在开发程序阶段最好使用允许范围检查指令, 在程序基本完成时禁止这一指令。在允许 R 指令时, 编译后的程序长度会明显加大, 执行也慢。使用时应该注意。

#### S 堆栈溢出检查

程序调用一个过程或者函数时, 是从局部变量堆栈中分配内存, {\$S+} 指令让 Turbo Pascal 加上必要的代码, 确保堆栈有足够的空间容纳这些变量。内存不够时, 程序将停止并产生一个运行错误。若使用 {\$S-} 指令, 当堆栈内存没有了时, 计算机将崩溃。堆栈检查要占用时间, 增加可执行代码, 因此只在开发调试阶段才使用堆栈检查指令。

#### V 字符串变量检查

当使用 {\$V+} 时, Turbo Pascal 对传递给过程和函数的字符串参数进行严格的检查。若不进行检查, 参数类型不匹配时, 有可能产生不可预测的结果, 如毁坏内存。进行字符串变量检查能发现隐蔽的字符串错误。禁止字符串变量检查要保证程序不会产生意想不到的破坏。

#### 参数指令

参数指令与开关指令不同, 没有明显的接通

关闭状态。这些指令表明编译时使用的文件名和程序所需内存的大小。

#### I 包含文件

在编译一个源程序时, I 指令使另外一个文件成为这个源程序的一部分。如果包含的文件没有指定扩展名, Turbo Pascal 假定为 .PAS。当一个程序太大, 不能一次放到 Turbo Pascal 编辑器中, 或者希望通过改变包含文件来修改程序的一部分时, 可以使用包含文件指令。例如:

```
INC. PAS 文件
Procedure Proc1
Begin
...
End
```

## 主程序文件

```
Program Main;  
{ $I Inc}      包含文件指令  
Begin  
Proc1;          执行包含文件  
...  
End
```

在这个例文中，编译程序把 Inc. PAS 作为主程序的一部分编译。

## L 链接目标文件

使用 L 指令可以把汇编语言目标程序与 Pascal 程序链接起来。这样可以在 Pascal 程序中使用汇编语言编写的子程序。这条指令是在 L 后面接目标程序文件名。

## M 内存分配

M 指令指示程序中堆栈和数组各自所用的内存量，这条指令的格式是在 M 后接三个数字，分别用逗号隔开。第一个表示堆栈所用的字节，第二和第三个表示数组的最小量和最大量。例如，{ \$M 10000, 300, 5000 } 表示分配给堆栈 10000 字节，分配给数组的最大字节为 5000，最小字节为 300。堆栈分配的内存量在 1024 到 65520 之间。数组分配的内存量为 0 到 655360 之间。

## O 覆盖单元名

O 指令的格式是在 O 后面接一个单元名。Turbo Pascal 将把这个单元包含在盖文件中。覆盖单元指令 O 必须用在 Uses 字句的后面。用 O 指令命名的单元必须用 { \$+ } 编译指令编译，否则不允许覆盖。

## 条件编译

使用条件编译，可以在一个文件中保留程序的不同版本。只要改变某些定义，就可以指示 Turbo Pascal 编译某些代码段，而把另外一些段隐藏起来，条件编译的关键是条件符号，即定义一个符号，控制条件编译的进程。下面的例子怎样进行条件编译。

```
Program Condtion;  
  
  { $DEFINE CON}                      (* 定义 CON *)  
  
  { $IFDEF CON}                       (* 如果 CON 已定义 *)  
  
    { $R+, S+ }  
  
  { $ELSE}                           (* 如果 CON 未定义 *)  
  
    { $R-, S- }  
  
  { $ENDIF}                          (* 条件编译结束 *)?  
  
Begin  
  
  { $UNDEFCON}                       (* 解除 CON 定义 *)  
  
  { $IFDEF CON}                       (* 如果 CON 已定义 *)  
  
  WriteLn('CON DEFINED');
```

{ \$ELSE }

( \* 如果 CON 未定义 \* )

WriteLn('CON NOT DEFINED');

{ \$ENDIF }

( \* 条件编译结束 \* )

程序在开始时定义了符号 CON。在第一个条件编译处，由于符号 CON 已经定义，则 R 指令和 S 指令都有效，而 R 指令和 S 指令禁止的语句被隐藏起来。在关键字 Begin 之后，CON 的定义解除，这时仅编译 WriteLn('CON NOT DEFINED'); 一行，而 WriteLn('CON DEFINED'); 不进行编译使用条件编译指令后，可以最大发挥 Turbo Pascal 的效率。下面介绍 Turbo Pascal 提供的条件编译指令。

**DEFINE** 定义

这条指令定义一个条件符号。所有依赖被定义符号的代码都将被编译，但只限于出现本指令的文件中的代码。例如，在一个主源程序文件，一个包含文件和一个单元文件中。在每个文件都使用条件编译指令。如果主源文件中使用了 DEFINE 指令，在包含文件和单元文件中不使用，则只影响主源文件，其它文件中不受主文件中 DEFINE 指令的影响。定义全局编译符号只能用 Options

Compiler 菜单上的条件定义功能。

**UNDEF** 解除定义

本指令与 DEFINE 指令相反。一个符号用 UNDEF 指令解除定义后，依赖于这个符号的代码都不编译。

**IFDEF**

如果条件符号已定义，本指令后面的代码将被编译。

**IFNDEF**

本指令与 IFDEF 相反，如果条件符号未定义，后面的代码将被编译。

**IFOPT**

本指令使得 Turbo Pascal 可以根据另外的编译指令控制编译。如 {IFOPTR+}，使编译程序在 R 指令有效时编译代码。

**ELSE**

作为 IFDEF，IFNDEF，和 IFOPT 的条件分枝，当条件为假时，编译本指令后面的代码。

**ENDIF**

本指令表示条件编译的结束，其后的任何代码都不依赖于条件符号。

### § 1.3.2 数据部分

在 Turbo Pascal 程序中，程序头和全局编译指令后是数据部分，包括全局的变量，常量，类型和标号。局部变量在过程和函数中说明，但它们遵守同样的基本规则。

**常量说明**

常量在以 CONST 开始的块中定义。在 Turbo Pascal 中有两种常量：无类型的和有类型的。无类型常量定义的格式为

CONST

<标识符>=<常量>;

常量可以是一个数值或字符及字符串。在标识符和等号之间加上类型定义,即为有类型常量的定义。下面是几个常量定义的例子:

CONST

WEEK=7;

Message='Hello! ';

TAX;Real=25.00;

Hours ; Integer=24;

MONTH:string[15]='september';

例子中前两个是无类型常量,后三个是有类型常量。对于一些在整个程序中不改变的值用常量标识符表示,这样简化了程序,也使程序易读,便于维护。

无类型常量和有类型常量在 Turbo Pascal 中是不同的。在编译程序时,每个无类型常量标识符都要用其值替换;而有类型常量标识符仅在数据段中定义一次,程序中使用有类型常量都是使用其副本。无类型常量要占用较多的内存。而有类型常量更象是一个已经初始化的变量,因此可以改变其值。如果要确保一个常量在程序中不变,应使用无类型常量。

#### 类型说明

在 Turbo Pascal 中可以定义自己的数据类型,然后用它们说明变量。类型定义在以 TYPE 开始的块中定义。类型定义的一般形式是。如:

type <标识符>=<数据类型>;

Type payday=(salary, Hours);

Employee=Record

    Name:String[20];

    Number:Integer;

    salary:Real;

End;

Maxstring=string[80];

NameList=Array[1..100]of string[20];

用户自己定义数据类型是 Pascal 的一大优点。在后面要做进一步的讨论。

#### 变量说明

变量是命名的内存区域。变量的说明在以 Var 开始的块中,变量说明的格式为 Var <标识符>;数据类型;

数据类型可以是 Turbo Pascal 标准数据类型,如布尔型,实型和整型,或在类型段中建立的用户定义的类型。例如:

Var

    i, j, R:Integer;

    x, y, z:Real;

    a, b, c:Boolean;

    Ad:payday;

    标号说明



标号标识程序中的一个位置，与 GOTO 语句共同使用，可以把控制从程序流中的一个位置强制跳到另外一个位置。对结构化语言来讲，使用 GOTO 语句不是一种好的方法。因此，Turbo Pascal 把标号的范围限制在同一过程块中。使用标号和 GOTO 语句还是有好处的，它可以增加程序的灵活性。特别是当需要从嵌套较深的循环中跳出时，使用 GOTO 语句不失为一种好的方法，否则将明显增加程序的复杂性。其格式为

Label

<标识符>, ... 标识符<>;

### § 1.3.3 代码部分

代码部分是 Turbo Pascal 程序中最大的部分，由使程序工作的指令组成。代码段包括一个程序块，由 Begin 开始，以 End 结束。在 Turbo Pascal 中，首先执行的程序模块是在程序的尾部。例如：

```
program exampla;  
  procedure Exa1;  
  Begin  
  End;  
  FuncTion Exa2; Integer;  
  Begin  
  End;  
Begin  
End
```

} 过程块

} 函数块

} 程序块

Pascal 被称为模块结构化语言，它的每一条语句都属于一个代码模块。模块可以嵌套，即在一个过程中嵌套另一个过程。简单的程序可以只有一个模块—程序模块。在一个过程中嵌套的过程对另一个过程中嵌套的过程是不可见的。这样在不同的过程中可以嵌套两个过程名相同的过程而不会行起混乱。过程名的作用域遵守以下规则：

- \* 一个程序可调用其所在块或嵌套于这个块中的任何子块中的过程。
- \* 当程序在较高层子块中说明了另一个具有相同名字的过程时，上一条规则无效。

在其它一些语言中，过程说明的顺序没有关系。但在 Turbo Pascal 中，一个过程未加说明之前不能调用（向前说明例外）。在 Turbo Pascal 程序中，最后到达程序块。程序块可能只有几个过程调用。因此程序可以看作一个连续进化的过程。程序的优先级是根据这样一个逻辑：由简单构造复杂。这虽然给程序中的过程规定了顺序，但有时可能要调用一个未说明的过程，这可以通过向前引用说明来实现。其格式为：

Procedure<过程名>; FORWARD;

FORWARD 是关键字。这里的过程只是一个过程头。整个过程在后面说明。这个指令通知编译器有一个前面未说明的过程。下面是一个向前引用过程的例子。

```
Program Prog1;  
Var  
  I: Integer;  
  procedure StepB(i: Integer); FORWARD;  
  Procedure StepA(i: Integer);
```

```

begin
  i:=i+1;
  StepB(i);
end;
procedure StepB;
begin
  writeLn(i);
  if i>100 then Halt;
  StepA;
end
begin
  i:=1;
  stepA(i);
End

```

在这个例子中 StepA 和 StepB 互相调用，这在过程优先级是不允许的，但用 StepB 的向前引用加以克服。

#### § 1.3.4 包含文件

Turbo Pascal 编译器只能容纳 62K 以内的正文。如果程序超过这个限制，必须把它们分成几块，分别放到多个文件中。在编译时，用包含文件指令把这些文件合并到一起进行编译。包含文件指令的格式为

(\$I <被包含文件名> \*)

当使用标准常用程序库时，包含文件指令是很有用的。

#### § 1.3.5 覆盖块

通过覆盖用户程序的单元，可以减少程序的内存需求量。为了支持覆盖，Turbo Pascal 提供了 OVERLAY 的单元使用覆盖的规则是：

- \* 打开 Force Far Calls 编译指令，编译程序和单元。
- \* 在程序的 Uses 语句中先对 OVERLAY 单元命名。
- \* 用 {\$O+} 编译指令编译覆盖单元。
- \* 用 {\$O<文件名>} 编译指令列出覆盖单元。
- \* 在执行任何语句包括初始化覆盖单元之前，先初始化 OVERLAY。

程序中覆盖处理相当简单，若覆盖单元中含有初始化代码，情况会稍稍复杂一点。Turbo Pascal 要求在程序的第一条语句执行之前就执行初始化代码。这意味着，在覆盖管理程序初始化之前就要执行覆盖单元，这个问题是这样解决的。不在主程序中初始化覆盖管理程序，而把启动代码放在一个单元中，并把这个单元放在 Uses 语句中，位于其它任何被覆盖单元之前。即要求 OVERLAY 单元作为 Uses 子句中的第一个单元。这样保证了在任何程序执行之前，覆盖管理程序已经安装完毕。

• Turbo Pascal 覆盖系统包括五个例程，用于初始化覆盖管理程序和控制程序使用内存。这些例程执行时设置全局变量 OverResult，用以指示是否发生了问题。覆盖单元定义下列常数

帮助理解 Overresult 的值。

const

OvrOK	= 0;	( * 无错误 * )
OvrError	= -1;	( * 非特定错误 * )
OvrNotFoond	= -2;	( * 覆盖文件未找到 * )
OvrNoMemory	= -3;	( * 内存不够 * )
OvrIOerror	= -4;	( * .OVR 文件读错误 * )
OvrNoEMSDriver	= -5;	( * 未装 EMS 驱动器 * )
OvrNOEMSMemory	= -6;	( * EMS 内存不够 * )

每次使用这五个例程中任何一个时, 用户程序都应仔细检查 OverResult 的值。下面分别介绍一下这五个例程:

OvrInit

这个过程初始化覆盖管理程序, 并准备好覆盖单元。OvrInit 带一个单字符串参数, 其中包含覆盖文件名。覆盖文件名通常与主程序文件名相同, 但带 .OVR 后缀。这个过程只能调用一次, 并且必须在任何覆盖单元之前调用。

OverInitEMS

覆盖尽管节省内存, 但是由于要频繁读磁盘, 大大减低了程序运行的速度。Turbo Pascal 覆盖管理程序可以把整个覆盖文件装到扩展内存中去, 这样可以大大减少访问覆盖过程的时间。只要执行 OvrInit EMS 过程, 就可以把覆盖文件装入扩展内存。在执行这一过程后, 如果计算机有足够可用的扩展内存, 覆盖文件将被装入, 否则, 仍要从磁盘上执行覆盖文件。

OvrSetBuf

在用 OvrInit 启动覆盖管理程序时, 得到运行覆盖过程所需的最小内存量。可以用 OverSetBuf 扩展这个内存。本例程带一个长整型参数, 表示以字节为单位的覆盖缓冲区的大小。这个缓冲区已必须大于最小缓冲区, 但小于 MemAvail。只有在数组没有动态变量时, 才有必要调用 OvesetBuf。

OvrGetBuf

这个函数返回表示当前覆盖缓冲区大小的值。在用 OvrSetBuf 增加缓冲区时, 可参考这个值。

OvrCLearBuf

这个函数清除覆盖缓冲区的所有覆盖单元。在执行了这个过程后, 调用覆盖过程必须要读磁盘。一般情况下不需要清除覆盖缓冲区, 只有在需要为其它目的重新使用覆盖缓冲区时, 才执行这个过程。

### § 1.3.6 过程与函数

模块化程序设计的思想就是把一个大的复杂问题分解成一些小而简单的问题, 每个小问题在程序中作为一个相对独立的部分, 在 Pascal 中这些部分称为过程和函数。由于过程和函

数相对独立,可以单独编写和调试,也比较容易维护。一个调试正确后,即可以加入到主程序中。

过程至少包括一个过程名和一个代码模块。其格式如下:

```
Procedure <过程名> (参数表;)  
Begin  
...  
End
```

} 代码模式。

只要调用过程名就可以在程序中执行过程。过程可以带参数也可以不带。不带参数的过程只能处理特定的在过程中出现的全局变量。带参数的过程通过参数传递可以处理任何类型相容的变量。

过程的参数传递有两种。一种是把变量的值。传递给过程,称为值参,这时对参数做任何改变都不会影响原变量,因为过程处理的只是原变量的临时副本,从过程中退出后,原变量不变。另一种传递给过程的不是原变量的值,而是变量在内存中的地址,称为变参。对变参做的任何改变都是永久性的,因为对变参所做传递给过程的实参共享同一个内存。退出过程后,改变依然存在。变参和值参在过程参数表中的说明是不同的。下面举例说明。

```
procedure ParaType(i:Integer) var j: Integer);
```

在这个例子中*i*是值参,*j*是变参。变参要用VAR说明,否则为值参。值参与变参的另一个区别是,值参可以是表达式,传递给过程的实参是表达式的值,变参是对应变量。

函数与过程相似,都是相对独立的代码块。函数定义的方式也与过程差不多。只不过函数用关键字FUNCTION,过程用关键字PROCEDURE。函数的格式为:

```
Function <函数名>(参数表):函数类型  
Begin  
...  
End
```

} 代码模块

除了定义的关键字不同外,函数与过程还有另外几个重要区别。首先,函数通过函数名回送一个结果值,在函数的参数表后面要说明函数的类型。过程的结果由参数回送,因此它可以不回送也可以回送多个结果。过程的参数表后面无过程类型说明。第二,函数体中至少要有有一个对函数名赋值的语句。而过程不能给过程名赋值,因此,过程体可以是空的。第三,过程用单独的语句调用,而函数调用必须出现在表达式中。此外函数一般只使用值参,也可以使用变参,说明方法与过程相同。

传递给过程或函数的参数与它们的形参在个数,顺序,类型上必须一致。但对串变量,可以用{\$V-}编译指令取消 Turbo Pascal 对串类型变参的严格检查,这时允许任意串类型变量作为任意串类型的参数。

Turbo Pascal 还可以使用另一类参数:无类型参数,无类型参数不对参数作类型说明。它的优点是可以传递任意类型的参数。定义类型参数时必须说明类型,因此 Turbo Pascal 很容易判别参数和变量类型是否匹配。对于无类型参数,过程和参数不知道要传递的参数的类型,这要由程序员自己处理。

所有的无类型参数都是变参,所以必须用VAR说明。过程不能直接使用无类型参数,因为过程不知道这个参数是什么。要使用这个参数必须在过程定义一个位于此参数的绝对变量,即这个变量与原参数位于同一地址。下面是一个无类型参数的例子:

```
Procedure Example(Var x);  
Var  
  y:Integer Absolute x;  
Begin  
  WriteLn(y);      (* 合法的, y 是整型数 *)  
  WriteLn(x);      (* 非法的; x 是无类型的 *)  
End;
```

其中变参 x 是无类型参数, y 是绝对变量。调用这个过程时,可以向它传递任何类型的参数,但过程将认为这个变量是整型数。如果变参不是整型时,将会产生完全不相关的值。但是在一些特定情况下,无类型参数还是很有用的。

## 第二章 数据类型与表达式

用计算机解决问题,就要处理各种各样的数据。任何程序都要对数据进行加工处理,不与数据打交道的程序可以说基本上没有什么意义。Turbo Pascal 为程序员提供了丰富的数据类型。

### § 2.1 标准数据类型

Turbo Pascal 为用户提供了一系列标准数据类型。Turbo Pascal 要求对每个变量都说明其数据类型。当类型不符时,会出现错误。

#### BYTE

一个 Byte 型变量占一个字节,为一个无符号值,范围在 0 到 255 之间。在算术表达式中,Byte 型变量可以由 Integer 或 LongInt 型变量赋值,只要后者的值在 Byte 型的范围内,

#### INTEGER

Integer 表示带符号的整型变量,即不带小数。占两个字节,范围是 -32768 到 32767。

#### WORD

与 Integer 型变量类似,但为无符号值,也占两个字节,范围从 0 到 65535。

#### LONGINT

LongInt 也是带符号的整型值。占四个字节,范围在 -2147,483,648 到 2,147,483,647 之间。上面的 4 种类型可以统称为整型数,它们都不能带小数点,及小数部分。它们运算速度较快,是最常用的数据类型。

#### REAL

对于整型不能处理的数据,Turbo Pascal 提供了 Real 型变量。Real 型有两种表示法:小数表示法和指数表示法。Real 型的数据范围是  $2.9 \times 10E-39$  到  $1.7 \times 10E38$  之间。

Real 型的算术运算速度比整型要慢得多,因为其结构复杂。Real 型变量遇到的另一个问题是溢出。当赋给 Real 型变量超出范围的值时发生溢出,并产生运行错误。但是情况并非如此简单。例如,当一个 Real 型变量的值为  $1.7 \times 10E38$ ,在这个变量上再加 1。这时变量的值并未发生变化,也不产生错误。在逻辑上应产生溢出,但检测不到。因为 Real 型变量只有 11 到 12 位有效数字。这样就会产生在大的 Real 型变量上加小的数值不改变原变量值的情况,此外,有时会出现不可预期的溢出。例如,一个值为  $1.7 \times 10E38$  的 Real 型变量乘以 1.0 时,变量的值不应改变,也不应产生溢出。但 Turbo Pascal 却检测到了溢出并停止运行。

#### CHAR

Char 型与 Byte 型一样占一个字节。但是 Char 型数据不能直接用于算术运算。Char 型数据主要用于正文处理,如文字比较和字符串赋值等。Char 型变量就是用 ASCII 码表示的字符,每个字符对应于一个特定的值。

#### STRING

String 型数据是存放正文信息的。它与前面几种变量不同,是以字符为单位的,而且其长

度不是固定的，可以在 1 到 255 个字符之间任意定义。为了表示长度，Turbo Pascal 规定字符串的首字节记录这一长度。所以 string 型变量的实际长度大一个字节。例如：Month:string[9] 则字符串 Month 长度为 9，但实际占用十个字节，因为第一个字节为字符串长度。当第一个字节中的字符串长度比字符串定义的长度小时，长度之后的字节在处理中将忽略。如 Month:='JUNE'，其存储格式为：

4JUNE????

第一个字节中的 4 为实际数值(二进制为 00000100)，第五个字节之后的字节 Turbo Pascal 不进行处理。维护字符串长度需要做一些额外的工作，因此字符串操作是 Turbo Pascal 所有操作中最慢的，在 Turbo Pascal 也可定义字符串而不指定长度，这时其长度为 255 个字符。

除上述数据类型外，Turbo Pascal 还支持另外几种数据类型。它们都属于 8087 数据类型，而且无论计算机是否安装 8087 数字协处理器，Turbo Pascal 仿真都可以存取。

SINGLE

这种类型是“短 Real”型，只占 4 个字节，只有 7 到 8 个有效位。

DOUBLE

这是“长 Real”，占 8 个字节，有 15 到 16 个有效位。

EXTENDED

这种类型是“非常长的 Real”，占十个字节。有 19 到 20 个有效位，数值范围为  $3.4 \times 10E - 4932$  到  $1.1 \times 10E132$ 。

COMP

这种类型不是浮点类型，占 8 个字节，有 19 到 20 个有效值，取值范围为  $-(2 \times 10E63) + 12 \times 10E63 - 1$ 。这种数据类型用于大数值运算。

## § 2.2 常量

Turbo Pascal 在程序启动时不对变量进行初始化，在对变量赋值之前，其值是未定的。常量在程序开始处可以由程序员赋值。程序中经常要用到的一些固定的数值，这对使用变量非常方便。当需要修改这些数值时，只要在定义处修改即可。不必在整个程序中一个一个地寻找修改。

Turbo Pascal 的常量有两种，类型常量和无类型常量。无类型常量是真正的常量，在程序运行中不允许改变其值。类型常量与变量相似，其值可以改变。两种常量在作为过程的参数时也有不同。两种常量都可作为值参，但只有类型常量才可作为变参。

## § 2.3 用户定义的数据类型

Turbo Pascal 的强大功能之一是可以由用户定义数据类型。根据程序的特定算法，可以定义相应的数据结构，可以增加程序的可读性，简化维护工作。用户定义的数据类型可分为三类，标量，记录和数组。

在定义数据类型时，最常用的两种方法是枚举和子界。枚举就是通过定义枚举类型并列出该类型可使用的枚举值来定义类型。在集合的例子中就曾用到枚举定义集合。枚举类型的一般定义如下

TYPE

<枚举类型标识符>=(<标识符>...<标识符>)

VAR

<枚举类型变量表>:<枚举类型标识符>为了定义枚举变量,可以首先定义枚举类型标识符,然后用枚举类型标识符定义枚举类型变量,枚举类型标识符在以 TYPE 开始的类型定义块中定义,在其中必须将该类型变量所允许的全部枚举值列在等号后的括号中,枚举值之间用逗号分开,每个枚举值在枚举类型定义中只能出现一次,并且只能在一个枚举类型定义中出现。

定义了枚举类型标识符后,就可以用来说明枚举类型变量。相同类型的几个枚举变量可列在枚举类型变量表中,变量之间用逗号分开,然后由一个枚举类型标识符说明。

另外,也可以不定义枚举类型标识符,直接在变量说明中给出它的枚举类型。即

VAR

<枚举类型变量表>:(<标识符>...<标识符>)

枚举类型的枚举值只能是标识符。枚举类型是有序集。第一个元素的值从 0 开始,以后按列举的顺序的分别给定其值。枚举变量的当前值可由 turbo Pascal 的标准函数 Ord 得到。枚举类型需占一个字节,最多可有 256 个元素。由于每个枚举值都有一个序号,可以把这个序号赋给数值变量,如:

VAR

i: Integer;

CoLor:(BlacK, Brown, Blue, Green, Red, Yellow, White);

Begin

CoLor:=Brown;

i:=ord(CoLor);

End

反之把数值变换成枚举量就不是这样简单了。如 CoLor:=i; 是非法语句。这个问题可用 Turbo Pascal 的标准函数 Fillchar 解决如

Begin

i:=1;

Fillchar (CoLor, 1, i);

End

Turbo Pascal 中可以定义子界类型数据。在定义了子界类型和子界类型变量后, Turbo Pascal 会自动检查变量是否超出了允许的范围。子界的一般定义为:

TYPE

<子界类型标识符>=<常量 1>..<常量 2>

VAR

<子界类型变量表>:<子界类型标识符>

子界类型决定该类型变量的取值范围,即该类型的所有变量必须在常量 1 和常量 2 之间。常量 1 和常量 2 必须属于同一个有序类型,如整型,字符型等等。常量 1 的次序数必须小于常量 2 的次序数。

由于枚举类型也是有序的,因此可以定义一个枚举类型的子界类型,但必须保证第一个



枚举值的序号小于第二个值的序号。在定义枚举类型子界前，必须首先定义枚举类型。下面是枚举类型子界的例子。

TYPE

Day = (Sundy, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);

Workday = Monday...Friday;

同样，也可以不预先定义子界类型，在变量类型说明中直接指出子界类型。即

Var

<子界类型变量表>; <常量 1> ... <常量 2>

## § 2.4 集合

集合是一种可以由用户定义的结构数据类型，它由一组相关的数或字符组成。集合可以进行通常的集合运算(并, 交, 差)，并可检查一个字符是否属于某一集合。集合类型一般如下定义：

TYPE

<集合类型标识符>; SET OF <基类型>

VAR

<集合类型变量表>; <集合类型标识符>

另外也可以如下定义：

<集合类型变量表>; SET OF <基类型>

集合定义中，基类型必须是有序类型，如整型子界，字符型，布尔型等。下面看几个例子：

Zero—Through—Nine; set of 0..9;

FullRange; Set of Byte;

UpperCase; set of 'A'..'Z';

Allchars; set of char;

第一个集合可包含 0~9 的整数的任意组合。第二个集合没有指定范围，只定义该集合可包含 0~255 (BYTE) 中的任何值。在后面的程序中可以把集合 FullRange 定义为使何数值子集。如：FullRange := [0..9];

这时 FullRange 与 Zero—Through—Nine 的元素相同。同样，第三个集合包含大写字母 'A' 到 'Z' 的任意组合。第四个集合为所有字符的组合，它也可包含从 0 到 255 这间的任何字符。

上面的第一和第二个集合是数值集合，第三和第四个集合称为字符集合。两种集合的最大范围都是从 0 到 255。但字符集合可直接与字符变量进行比较。

另外一种集合可以由用户定义其元素。这些元素可以不是字符，也不是数值。但在定义中必须一一列出，其元素的最大个数不得超过 255 个。例如：

TYPE

MONTH; Set of (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC)

VAR

WINTER,SUMMER;MONTH

Begin

WINTER:=[DEC, JAN, FEB];

SUMMER:=[JUN..AUG];

集合最多可用 32 个字节存贮, 等于 256 个不同的分位。如果集合的元素少, 可以用较少的字节, 集合可以自动减少所用字节。

## § 2.5 数组

数组是将几个数据用一个单独的名字处理, 不论是标准数据类型还是用户定义的数据类型都可扩展为数组。同一数组中的每个元素用下标区分, 数组的一般定义形式如下:

Type

<数组类型标识符>=Array[下限…上限]of <数据类型>;

Var

<数组变量表>:<数组类型标识符>;

下标可以是任何有序类型, 如整型, 字符型的及枚举类型或子界类型。数组的另一种定义格式为:

<数组类型标识符>:Array[下限…上限]of<数据类型>; 数组下标的下限序号必须小于上限。数据类型表示数组中每个元素所具有的类型, 这个数据类型可以是 Turbo Pascal 的使何标准类型, 或用户定义的类型, 包括数组和其它结构类型。

引用数组中的任一个元素可以通过在数组变量名后加括在方括号中的下标来实现。下标可以是常量, 变量或表达式, 表达式的类型必须与说明中的下标类型一致。下面举个数组的例子:

Price:Array[1..365] of real;

Price[10]:=34.50;

其中第一行定义了一个有 365 个元素的数组, 数组中的每个元素都是实型的。第二行对这个数组中的第 10 个元素赋值为 34.50。

数组可以作为过程或函数的参数传递。作参数传递时。实参数组必须与形参数组类型相同。两个数组具有相同的结构, 即数据类型与下标类型都相同, 才作为类型相同处理。

数组也可以有不止一个的下标。这样的数组称为多维数组。常用的是二维数组, 有时也称二维数组为矩阵, 多维数组的一般定义方式如下:

Type

<数组类型标识符>=Array [下限 1..上限 1] of Array[下限 2..上限 2]  
of..Array[下限 n..上限 n]  
of 数据类型;

或者

Type

<数组类型标识符>=Array [下限 1..上限 1, 下限 2..上限 2,  
.., 下限 n..上限 n]of 数据类型;

第三种方法是在变量说明中给出。其格式为:

<数组变量标识符>:Array [下限 1..上限 1, 下限 2..上限 2,  
..., 下限 n..上限 n]of 数据类型。

下面例子说明什么是多维数组。

```
Type  
Name=Array[1..10]of char;  
Var  
Student;Array[1..100] of Name;
```

这个例子中, Name 是一个有十个元素的一维数组。Student 是有 100 个元素的数组, 每个元素是一个有 10 个元素的数组。因此 student 是一个二维数组。下面的两个数组与上面的数组相同, 即为多维数组的另外两种定义格式。

```
Student;Array[1..100] of Array [1..10] of Char;  
student;Array[1..100 , 1..10] of char;
```

当对数元素赋值时, 只要指出相应下标即可。如:

```
Student[10, 5]; ='A';
```

在多维数组中, 很难记住某一行中存放的是什么类型的值。因为数组没有提供任何名字。为此最好对变量使用有意义的名字。Turbo Pascal 为多维数组提供了一种替代物: 记录数组。下面的例子说明了记录数组为什么比多维数组好用:

```
Observation ;Array[1..2200] of record
```

```
    Temperature;Real;
```

```
    Conductivity;Real;
```

记录数组的定义很明确, 它由一系列温度和导电率的观测值组成。它们可以通过名字以下面方式引用:

```
Observation[1]. Temperature
```

```
Observation[1]. conductivity
```

所有使用多维数组的地方, 都可以考虑用记录数组代替, 这样可以使程序更清晰。

## § 2.6 记录

在 Turbo Pascal 中使用记录, 使得数据的组织和表示更加方便灵活。使用记录有两个优点: 首先, 单个记录的每个元素在逻辑上是互相关联的, 这有助于记忆。第二, 某些操作如赋值和文件操作可以对整个记录进行, 不必引用其中的每个元素。

象数组一样, 记录是两个或两个以上有关数据的汇集。但是记录的各个分量可以有不同的类型。如可用一个记录存贮一个人的各项数据: 名字(字符串), 婚姻状况(枚举), 年龄(整型), 出生日期(记录型)。

在使用一个记录之前, 必须先定义记录的类型。在类型说明中指出记录类型的名字和其中每个域的名字及类型。记录类型的一般定义形式为:

```
Type  
<记录类型标识符>=Record  
    <域标识符表>:<类型>;  
    ... ..
```

<域标识符表>; <类型>;

End;

其中 Record 和 End 是保留字, 不能缺少。每一行的域标识符表可以是一或多个标识符。若标识符不止一个, 每个标识符之间用逗号分开。类型可以是任何标准数据类型或用户定义的数据类型, 包括数组或另外一个记录类型。对于用户定义的数据类型可以在记录之前定义, 也可以在记录类型中直接说明其类型。

定义记录类型的方法是在变量定义中直接定义其类型, 其形式为:

Var

<记录变量标识符>; Record

<域标识符表>; <类型>;

... ..

End;

记录的使用很简单, 引用方法有两种: 显式引用和用 With 保留字隐含引用。一般情况下, 按照某一特定条件处理记录的每一个域。使用域选择可以引用记录某个的域, 方法是在记录名和域名之间用圆点隔开。即

<记录名>. <域名>

如果每次引用记录的域时, 都使用显示引用, 会使程序显得十分冗长。使用保留字 With 后, 在引用记录的域时不必写出记录名, 即隐式引用。With 语句的一般形式是:

With <记录变量名> Do

<语句体>

语句体可以是单独一条语句, 也可以是若干条语句。如果是一条以上的语句, 语句体应以 Begin 开始, 以 End 结束。在语句体中引用记录的某个域时, 只需使用域名, 不必使用记录名, 下面举个例子说明记录的使用。

Program X;

Var

Rec1, Rec2; Record

a: string[20];

b: Integer;

c: real;

End;

Begin

Rec1.a := 'sss';

Rec1.b := 1;

Rec1.c := 12.23;

With Rec1 Do

Begin

a := 'sss';

b := 1;

c := 12.23; End;

End.

在这个例子中, 前三个赋值语句和后三个赋值语句的作用是相同的。前者是显式语句,

在这个例子中，前三个赋值语句和后三个赋值语句是作用相同的。前者是显式语句，后者是隐含引用。使用 With 的隐含语句可以嵌套，这样，用 With 语句可以引用多个记录。如

```
With Rec2, rec1 D.
```

可以隐含引用两个记录。但这也可能出现问题。比例如上面的例子中都有记录域 a，这时使用赋值语句

```
a := 'sss';
```

是对哪个记录进行赋值就成为问题。因为这种引用没有告诉 Turbo Pascal 引用哪个记录。这时编译程序认为歧义元素属于含该元素“最近”的记录。因而上述语句中认为 a 属于 Rec1。即与下述语句相同。

```
Rec1.a := 'sss';
```

对于记录可以用一条语句把一个记录的所有元素赋给另一个记录。如

```
Rec2 := rec1;
```

把 Rec1 的所有元素对应地赋值给 rec2 的元素，记录的某个域也是记录时，称为层次记录。下面用例子说明层次记录。例：

Type

```
Alfa = Packed Array[1..15] of char
```

```
Month = (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC);
```

```
Months = Array[1..12] of Month;
```

```
Address = Record
```

```
    City: Alfa;
```

```
    streetName: Alfa;
```

```
    streetNumber: Integer;
```

```
End;
```

```
Date = Record
```

```
    MonthName: Month;
```

```
    Day: 1..31;
```

```
    Year: 1990..1999;
```

```
End;
```

```
Employee = Record
```

```
    Name: Alfa;
```

```
    Number: Integer;
```

```
    BirthDate: Date;
```

```
    Home: Address';          salary: Real;
```

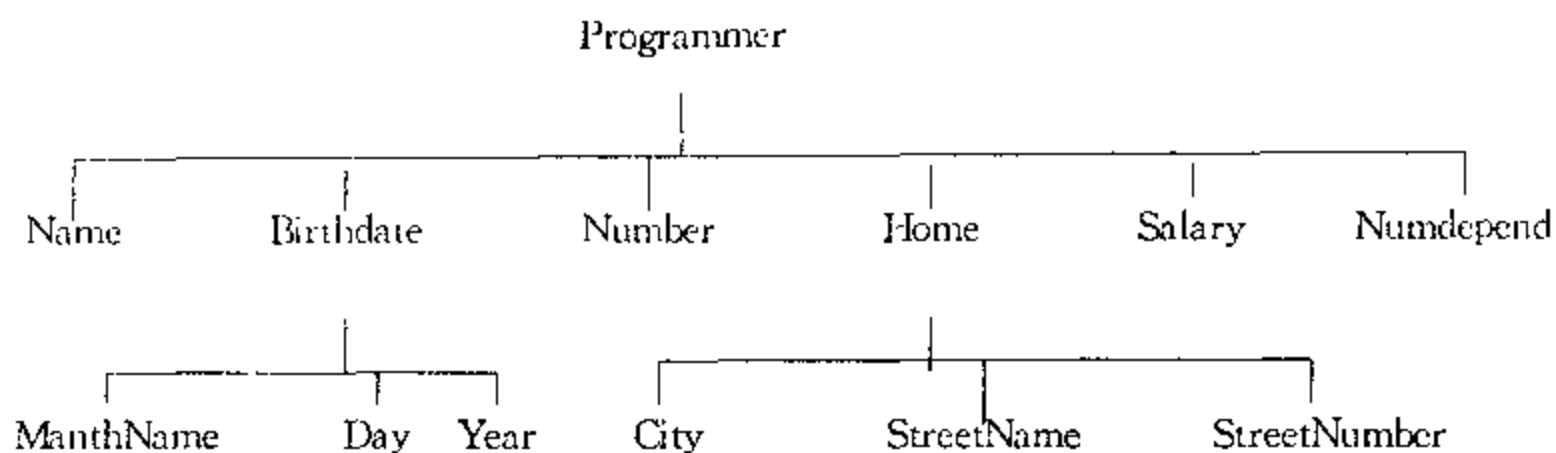
```
    Numdepend: Integer;
```

```
End;
```

```
Var
```

```
    programmer: Employee;
```

在这个例子中，记录变量 programmer 的层次结构为：



记录变量共有 6 个元素，其中两个元素 Birthdate 和 Home 也是记录。在这种情况下，引用记录的某个元素时，必须使用自顶向下的完全路径。如 programmer, Birthdate, year 是引用子记录 programmer, Birthdate 的元素 Year，但是 programmer, year 是不完全的，并且无定义。在这种情况下也可以用 with 语句隐含引用。如

```
with Programmer, Birthdate D
```

```
Writeln('YEAR OF BIRTH=', Year, ' Day of BIRTH=', Day);
```

输出记录 programmer, Birthdate 中的两个元素 Year 和 Day。

也可以使用 With 的嵌套。如

```
with Programmer, Birthdate, Home D.
```

```
Writeln(Name, 'LIVES IN' City, 'AND WAS BORN IN ' Year);
```

将输出记录 programmer 中的 Name，子记录 programmer, Home 中的元素 City，子记录 programmer, Birthdate 的一个元素 Year。它与下面的 With 语句等价。

```
With programmer D.
```

```
With Birthdate D.
```

```
With Home D.
```

```
Writeln(...);
```

因此在 With 语句中，记录名出现的次序是很重要的。从第一个到最后一个分别对应最外层和最里层。在引用元素时，完全域的选择是从里层向外层逐层匹配直到最外层，这样得到完全路径。因此，对于层次记录使用 With 语句时应首先写记录变量名，后写子记录变量名。

前面的所有同一记录类型的记录变量都具有同样的结构和形式。Turbo Pascal 允许使用另外一种记录：变体记录。变体记录中某些元素在不同的变量中可以是不同的，即变体部分。记录的变体部分用保留字 Case 开始，放在记录的固定部分之后。变体记录一般形式如下：

Type

<记录类型标识符> = Record

<域名表> : <类型>;

...

<域名表> : <类型>;

case <标志域> : <类型> of

<常量表> : (<域表>);

... ..

— 固定部分

— 变体部分。

标志域的类型必须是前面定义的枚举或子界类型。若某个常量表包括几个常量时，几个常量之间用逗号分开。域表列出与常量表相对应的域名和类型。域表中的每个域说明用逗号分开，域表用括号括上，在定义变体部分时应注意以下几点。首先，所有域名必须各不相同。一个域名不能既出现在固定部分，又出现在变体部分，也不能同时出现在变体部分的两个域表中。第二，如果某常量表没有对应的域，这时域表为空，用一对空括号表示。第三，域表中若也含有变体部分，则它必须在域表中的固定部分之后。此外，Case 开始的变体部分不需要单独的 End 与之匹配。尽管域名必须各不相同，但允许几个域说明为同一个记录类型。下面的程序使用了变体记录

```

program Variantecord;
Uses CRT;
Type
  VehicleType = (Car, Boat, plane);
  VehicleRec = Record
    IDnumber: Integer;
    Price: Real;
    Weight: Real;
    Case kind: VehicleType of
      Car: (MilesperGallon: Integer;
        Odometer: Real);
      Boat: (Displacement: Real;
        Length: Integer);
      Plane: (Engines: Integer;
        seat: Integer);
    End;
  Var
    Vehicle: VehicleRec;
Begin
  Clrscr;
  Vehicle.IDnumber := 123;
  Vehicle.price := 12000;
  Vehicle.Weight := 1200;
  Vehicle.MilesperGallon := 21;
  Vehicle.Odometer := 75000.0;
  With Vehicle DO
    Begin
      Case kind of
        Car;
        Begin
          Writeln('Kind = Car');
          Writeln('Miles per gallon = ', Milespergallon);

```

```

        WriteLn('Odometer=', Odometer;0:1);
    End;
Boat;
    begin
        WriteLn('Kind=Boat');
        WriteLn('Displacement=', Displacement);
        WriteLn('Length=', Length);
    End;
plane;
    Begin
        WriteLn('Kind=plane');
        WriteLn('Engines=', Engines);
        WriteLn('Seats=', seats);
    End;
End;
WritLn;
write('press Enter...');
Readln;
End.

```

在上面例子中, 域 Car, Boat 和 plane 中共有 4 个整型和两个实型, 共 20 个字节。不过由于记录的变体部分共享同一块内存, 这个记录的变体部分一个单块最多只需 8 个字节(一个整型和一个实型)。

标志域 Kind 指明使用哪部分变体记录。当标志域使用后, 变体部分就指明了使用哪种形式。

另一种变体记录不使用标志域。称为自由结合。这种自由结合变体记录很象 Cobol 中的重定义域。下面的程序说明如何使用自由结合变体记录。

```

Program freeVnion;
Uses CRT;
Type
    CharByte=Record
        Case Integer of
            1: (Characters:Array[1..10] of char );
            2: (Number:Array[1..10] of Byte);
        End;
Var
    CB:CharByte;
Begin
    Clrscr;
    With CB D.
        Characters[1]:='A';

```



With CB D.

WriteLn('Numeric Value of Character A is 'Numbers[1]);

WriteLn;

Write ('Press ENTER...');

ReadLn;

End.

在这个例子中, 变体记录中没有标志域, 只有一个数据类型 `kk` 整型。标志域空缺意味着不需存贮标志值, 可以不受限制地引用变体元素。这个程序很有意思, 因为变体记录用两种不同的方式定义了一个数组。首先定义了一个 10 字节的字符数组, 在下一行又把它定义成字节数组。由于是变体记录, 这两个数组共享同一块内存。这个数组中的元素既可以作为字符利用, 也可以作为数字引用。这在程序中作了演示。其中首先使用标识符 `Characters` 对数组的第一个元素赋了一个字符, 然后用标识符 `Numbers` 把这个元素作为数字输出。

## § 2.7 Turbo Pascal 中的表达式

计算机主要用于计算和数据处理。Turbo Pascal 提供了强大的算术函数和扩展逻辑命令。

### § 2.7.1 算术运算

算术运算是基于表达式和赋值语句。赋值语句是 Turbo Pascal 中最常用的语句。它的一般形式是:

`<变量> := <表达式>`

它的定义是把表达式的值放入变量。

表达式可以是常量, 变量, 数值、函数及操作符组成, 其结果是特定的值。如 `4+3` 是表达式, 但不是 Turbo Pascal 的语句。表达式必须出现在赋值语句中。如

`R := 4+3;`

或者出现在逻辑语句中, 如

`IF R = (4+3) Then ...`

这两条语句虽然表面上差不多, 但是意义却不同。在赋值语句中是把 `4+3` 的结果 7 放入变量 `R` 中。在逻辑语句中, 是把变量 `R` 的值与表达式的结果 7 进行比较, 如果 `R` 的值是 7 则产生一个布尔变量 `True`, 否则产生 `False`。操作符 `:=` 是赋值, `=` 是相等比较。在赋值语句中不含相等概念。如 `x := x+1` 在数字上是错误的, 但在 Pascal 中却是合法的, 因为它是将变量 `x` 的值加 1 再赋给 `x`。

Turbo Pascal 的算术运算主要用于整型数和实型数。结果也是整型数和实型数。表达式是整型数有两个条件: 首先表达式的所有操作数是整型数整型变量或整型化的其它变量。第二, 如果进行除法, 必须用 `DIV` 代替

。其它算术表达式都为实型数。操作数中只要有一个实型数, 则结果也是实型数。

Turbo Pascal 的算术表达式与通常的数字表达式稍有不同。所有的操作数, 算符都应写在一行上。操作数必须是合法的标识符或数值。乘法用 `*` 表示, 且不得省略。算术运算分四个级别:

第一级    `+, -`    (单目)

第二级    ( )    (括号)

第三级    \* , / , DIV ,

第四级    + , -

表达式中的运算按优先级从高向低进行。同一优先级和运算按从左到右的次序进行。最高级单目算符 + , - , 表示数的正、负。可以按下列方式使用。

$R = 1 - -3;$

最后 R 中的值为 4。

第二级是括号, Turbo Pascal 中只使用圆括号 ( )。括号中的运算比括号外的运算先进行。使用括号可以使表达式更清晰。

第三级乘法和除法与第四级是加法和减法。由于 Turbo Pascal 的强类型, 这些运算对整型和实型数是不同的。

当表达式中的操作数中的浮点数或除法使用算符

时, 表达式的值是实型的。整型表达式可对实型变量赋值, Turbo pascal 隐含地将整型值转换为实型值。如

```
Var
i, j: Integer;
X : real;
Begin
i:=100;
j:=150;
x:=i*j;
End
```

在这个例子中整型表达式  $i*j$  向实型变量  $x$  赋值。在转换成浮点之前, 表达式按整型运算。这可能产生错误; 整型数的上限为 32767。如果整型数太大, 整型表达式转换成浮点之前就可能溢出, 产生错误的结果。如

```
i:=10000;
j:=10000;
x:=i*j;
```

则  $x$  的值为 -7936。显然是错误的。为了减少这种错误, 必须在表式中加上一个浮点数, 使整型表达式变为浮点表达式。如:

```
x:=1.0*i*j;
```

当表达式赋值给整型变量时, 应使用整型运算。整型表达式的所有操作数必须是整型或转换为整型的实型。把实型转换为整型号以使用两个 Turbo Pascal 的标准函数 Round 和 Trunc。这两个函数都可以把实型值转换为整型值, 但是转换方式稍有不同。Round 函数是把实型数的小数部分作四舍五入转换, 如  $\text{round}(10.49)$  的结果是 10, 而  $\text{Round}(10.5)$  的结果是 11。

而 Trunc 是把小数部分舍去。如  $\text{Trunc}(10.9)$  的结果是 10。利用这两个函数, 可以在整型表达式中使用实型变量。但是应注意, 实型的值不要超过整型值的范围。

## § 2.7.2 整型运算

Turbo Pascal 中有一些运算专门用于整型。这些操作在实型中没有相应的运算。除前面介绍过的整数除 DIV 外,对整型数还有取模运算 MOD。这个运算是取除法的余数。如:7MOD2 的结果是 1,而 7DIV2 的结果是 3。

除了上面两个运算外,Turbo Pascal 还有其它专用于整型的运算。如:And, Or, Xor, Shl, Shr 等。这几个运算都可以作用于一个整型值的特定位。因而可称为位运算。

数在计算机内都是按二进制存贮的。一个字节占 8 位,每一位可放一个 0 或 1。一个整型数占两个字节。一个整型数最大为 65535,按二进制表示为 11111111 11111111b。但是在 Turbo Pascal 中,最高位是符号位。0 表示正数,1 表示负数。因此,Turbo Pascal 中最大整数值为 32767,或 01111111 11111111b,最小整数是-32768,或 11111111 11111111b。

And, Or, xor 都是按位运算。And 是“与”运算,Or 是“或”运算,Xor 是“异或”运算,也称为“模二加”。运算结果如下:

And	0	1	or	0	1	xor	0	1
0	0	0	0	0	1	0	0	1
1	0	1	1	1	1	1	1	0

即“与”运算:1 And 1 为 1,只要操作数中有一个 0,结果即为 0。“或”运算:0 Or 0 为 0,只要有一个操作数为 1,结果即为 1。“异或”运算:两个操作数相同时结果为 0,不同时结果为 1。

运算 Shl 和 Shr 是移位操作。Shl 是左移,Shr 是右移。一个字节向左或向右最多移八位,这时所有的位都为 0,当向左移一位时,所有的位都向左移一位,最高位移出丢失,最低位补一个 0。右移时类似,只是向右移,最低位移出丢失,最高位补一个 0,如

10110101

左移一位 01101010

## § 2.7.3 算术函数

Turbo Pascal 提供大量的标准算术函数,可以用它们简化计算。下面做一简要介绍。

Abs(num) 返回 num 的绝对值。传递的值可以是整型,也可以是实型,返回的值与传递的值的类型相同,如果 num 是整型,Abs() 返回的是整型。

Arctan(num) 返回 num 的正切值。num 可以是整型也可以是实型。返回的值是实型。

Cos(num) 返回 num 的余弦值。num 可以是整型,也可以是实型。返回的值是实型。

Exp(num) 返回自然数的 num 次幂。num 可以是实型,也可以是整型。返回的值是实型。

如果使用标准实型(非仿真 8087),当 num 大于 88 或小于-88 时,产生溢出错误。

Frac(num) 返回 num 的小数部分。num 可以是实型或整型。如果 num 为整型,返回的值为 0。结果是实型。如果使用标准实型(非仿真 8087),大于 1.0E10 的数,返回值都为 0。

Hi(num) 返回一个整数,其高字节为 0,低字节为 num 的高字节。num 必须为整型。

Int(num) 返回 num 的整数部分。num 可以是实型，也可以是整型。如果 num 是整型，这个函数不改变其值，只是把它变成整型。

Ln(num) 计算 num 的自然对数。num 可以是实型 也可以是整型，但必须大于 0。

Lo(num) 返回一个整数，其高字节为 0，低字节为 num 的低字节。num 必须为整型。

Ord(num) 返回任何是标量的相对值，包括字符型。结果是整型。

Pre(num) 返回 num 减 1 的值，num 为整型，结果也是整型。

Random 返回 0 和 1 之间的随机数。结果是实型。

Random(num) 返回一个 0 和 num 之间的随机数。num 是整型的，结果也是整型。

Round(num) 返回最接近 num 的整数值。num 是实型，结果是整型。

Sin(num) 返回 num 的平方。num 是实型或整型；结果是实型。当使用标准实型(非模拟 8087)，如果 num 超过 1.0E18，将产生溢出错误。

Sqr(num) 返回 num 的平方。num 为实型或整型；结果向实型，在使用标准实型(非模拟 8087)时，如果 num 超过 1.0E18，将产生溢出

Sqr(num) 计算 num 的平方根。num 是实型或整型；结果是实型。

Succ(num) 返回 num 加 1 的值。num 是整型。结果是整型。

Trunc(num) 返回 num 去掉小数部分的值。num 是实型，结果是整型。

#### § 2.7.4 逻辑运算

逻辑运算通常用于条件测试。Turbo Pascal 提供了下列逻辑运算符：

=	等于
<>	不等于
<	小于
>	大于
<=	小于等于
>=	大于等于
Not	变反
Case	多重比较

逻辑表达式与 If—Then 一起可以决定程序应执行哪些语句。If—Then 可以控制一条语句的执行，也可以控制由 Begin—End 界定的代码块。如果 If—Then 再与 Else 配合，可以控制分枝程序块的执行、即当 If 后的条件满足时，执行 Then 后面的代码块，否则执行 Else 后面的代码块。这里应注意的一个问题是，Else 前面的一条语句不以分号结束，因为 Turbo Pascal 认为 Else 是一条长语句的延续。例如

```
If a>b Then
  Begin
    WriteLn('A is greater than B');
    b:=a;
  End
```

```

Else
Begin
WriteLn('A is not greater than B');
a:=b;
End;

```

用 If-then 语句的多 Else 结构, 可以产生多条件分枝。如:

```

If a=1 then
Begin
WriteLn('A equals 1');
End
Else if a=2 Then
Begin
WriteLn('A equals 2');
End
Else if a=3 Then
Begin
WriteLn('A equals 3');
End
else if a=4 then
Begin
WriteLn('A equals 4');
End;

```

在多条件分枝选择的情况下, 使用 Case 语句更简明易读, 更具灵活性。如:

```

Case a of
1: WriteLn('a equals 1');
2..4: begin
WriteLn('a is between 2 and 4');
WriteLn('case statements can specify ranges ');
End;
5: WriteLn('a is equals 5');
else Begin
WriteLn('a is not between 1 and 5');
writeLn('The case statement supports the else clause');
End;
End;

```

case 语句允许定义值的范围, 如上例中的 2..4。但是 case 语句只允许使用简单数据类型, 这比 If-Then 语句严格。If-Then 语句可以使用实型和字符串, 但 case 语句不允许使用。

### § 2.7.5 集合运算

集合是 Pascal 中的一个有特色的结构数据类型。对 Pascal 中的集合可以执行通常的集合

运算。在处理集合之前，先要给集合变量赋初值。集合的赋值语句左端为集合变量，右端为一个集合表达式。集合表达式可以是一个集合值，由元素枚举表或子界指出，一个集合变量，或几个集合变量的运算。在赋值语句中，右端的集合表达式必须与左端的集合变量相容。

有时，可能需要使一个集合为空。在 Pascal 中，空集用一对方括号[]表示。如

Day := [];

集合的并，交，差运算，要求运算的两个集合类型相容。两个集合的并集是两个集合中所有元素的汇集。如

A := [1, 3, 4];

B := [1, 2, 4];

C := A + B;

则 C 为 [1, 2, 3, 4]。两个集合的交集是两个集合中所共有的元素的汇集。如

C := A \* B;

则 C 为 [1, 4]。集合 A 与集合 B 的差集是在集合 A 中但不在集合 B 中的元素的汇集。如

C := A - B;

则 C 为 [3]。

在集合中，元素的次序无关紧要。只要两个集合中的元素相同，次序无论怎样排列都认为是相同的。在集合表达式中，可以有多个集合参加运算，在运算中遵守通常的优先规则：先交(\*)，后并(+)，差(-)，同级运算从左至右进行。如希望改变求值顺序可使用括号。

集合也可以进行逻辑运算。逻辑运算的对象必须是相容的集合。结果为布尔型变量。运算相等=，不等<>可用于检查两个集合是否包含同样的元素。如

[1, 3] = [1, 3] 是 True

[1, 3] <> [3, 1] 是 False

集合还有子集与包集的关系。如果集合 A 的所有元素都是集合 B 的元素，则集合 A 称为集合 B 的子集，如果集合 B 的所有元素都是集合 A 的元素，则称集合 A 是集合 B 的包集。即  $A \supset B$ ，下面是子集与包集的一些例子。

[1, 3] ≤ [1, 2, 3, 4]                      为 True

[1, 4] ≤ [1, 4]                            为 true

[1, 2, 3, 4] ≥ [1, 4]                      为 true

[1, 3] ≥ [1, 3, 4]                        为 false

除了上述运算外，集合运算中还有一个特殊的新运算 ln，用于确定一个元素是否包含在某个集合中，即集合成员检查，其它的运算要求参加运算的两个对象都是集合。对于这些运算容易犯的一个错误是，单元素集合不用方括号起来。如：

[1, 3, 4] + 2

是错误的，正确的形式为：

[1, 3, 4] + [2]

但对于集合成员检查运算 ln，参加运算的两个对象一个是元素，另一个是集合。元素必须与集合的数据类型相容，如：

1    `ln[1,2,3,4]`    为 True

1    `ln[2,4]`    为 False

用 `ln` 运算可以把逻辑表达式写得更明确自然。如：

`(CH>='0')And (CH<='9')`

可写成：

`CH ln ['0'..'9']`

同样下式：

`(CH>='A') And (CH<='z')`

可以写成：

`CH ln ['A'..'z']`

一般来讲，集合都可以用数组表示，但有些运算用集合比数组更有效。如：

`Operator := Operator - ['*'];`

可以从集合 `Operator` 中去掉元素 `'*'`。如果 `Operator` 不是集合是数组，要从中去掉元素 `'*'`，首先要确定 `'*'` 在数组中的位置，然后用某种方法去掉它。这显然比写一个赋值语句复杂得多。

## § 2.8 类型间的关系

Turbo Pascal 中有各种各样的数据类型，它们之间的运算必须遵照一定的关系。数据类型之间有三种关系：相同，相容，赋值相容。

若满足下列条件之一，称变量为类型相同。相同关系是对称的，即若 A 与 B 相同，则 B 与 A 也相同。

- \* 使用同样的类型标识符说明。
- \* 类型标识符不同，但用形如 `T1=T2` 的说明定义为等价。
- \* 在同一条变量说明语句中说明。如 `A,B; 1..2`。

满足下列条件之一，则称变量的类型是相容的。相容关系也是对称的。

- \* 是相同的非结构类型。
- \* 为相同类型的子界，或一个是另一个类型的子界。
- \* 为具有相同数据类型的集合类型。
- \* 为长度相同的串类型。
- \* 整型与实型

满足下列条件之一，则称表达式 E 的类型对变量 T 的类型赋值相容。赋值相容是不对称的。

- \* E 与 T 为类型相同的非文件类型。
- \* T 为实型，E 为整型或整型的子界。
- \* E 与 T 为相容的简单类型，且 E 的值是 T 所允许的。
- \* E 与 T 是相容的集合类型。且 E 的每个元素都是 T 的类型所允许的值。
- \* E 和 T 是相容的串类型。

类型间的相同，相容，赋值相容是三个完全不同的概念，它们之间不存在简单的蕴含关系。下面举例说明：

Type

T1=File of Real;

T2 = 1..100;

T3 = 10..20;

T4= Array[1..20] of Real;

var

A1, a2:T1

A3:T2

A4:T3;

A5, A6: T4;

在这个例子中 A1 与 A2 类型相同，都是文件类型，但它们不相容，也不赋值相容，因此，由类型相同得不出相容或赋值相容的关系。这里有一个新的变量类型—文件类型。这个类型将在后面介绍。同样，由相容关系也得不出相同或赋值相容的结论。如 A3 与 A4 类型相容。但类型不相同，因为超界，A3 对 A4 不赋值相容。赋值相容可能不相容，如 A5 与 A6 是相同的，而且赋值相容的数组类型，但它们不相容。同样，赋值相容也可能不相同。如 A3 与 A4 类型相容，A4 对 A3 赋值相容，但不相同。

在 pascal 中，不同的地方要求的类型之间的不同关系。否则将导致错误。在算术运算，逻辑运算，关系运算与集合运算时，要求参加运算的对象类型相容。型及其子界与实型是相容的，因此整型与实型可以进行混合运算、相容的集合可以进行并，交，差运算及关系运算。相容的字符串也可以进行关系运算(比较运算)。

记录与数组没有相容性。所以数组与数组之间，记录与记录这间不能进行上述运算。但对于数组元素和记录的分量，如果相容可以进行上述运算。

在赋值语句，Read 语句和 For 语句中要求赋值相容。在赋值语句中，等号右边的表达式必须与左边的变量类型赋值相容。变量为实型的，则允许表达式为实型或整型的。如果变量为整型，表达式必须为整型。相容的子界或相容的集合在赋值时，要保证表达式的值不要超过左边变量所允许的范围，否则会引起赋值不相容。

只要数组类型赋值相容(即类型相同)，可以把一个数组整体赋值到另一个数组。同样，如果赋值相容，一个记录可以整体赋值到另一个记录。但文件不具有赋值相容，因此不能将一个文件整体赋值到另一个文件中。

在 Read(V)语句中，要求输入的数据与变量 V 的类型赋值相容。当 V 为实型变量时，可以输入实型数或整型数。若 V 为整型变量，则只能输入整型数。

对于 For 语句，For<循环变量>:=<初值>TO<终值>DO 中，要求初值、终值与循环变量类型赋值相容，而且必须是有序类型。

在过程与函数调用中，如果是变参，要求实参与其对应的形参类型相同。如果是值参，则要求实参与对应的形参类型赋值相容。过程与函数的值参和变参只能用类型标识符说明，为了保证实参与变参类型相同，也必须用同一类型标识符，或与其等价的类型标识符来说明。



## 第三章 程序的控制结构

计算机程序可分为三种基本结构：顺序结构，选择结构，循环结构，顺序结构是最简单的结构，它从 Begin 开始，顺序执行每一条语句，直到最后的 End 语句，下面是一个顺序结构程序的例子

```
Program Example;  
Uses CRT;  
Var  
    i,j,k:Integer;  
Begin  
    Clrscr;  
    WriteLn('Enter first Number:');  
    ReadLn(i);  
    WriteLn('Enter Second Number:');  
    ReadLn(u);  
    K:=i+j;  
    WriteLn('Sum=',K);  
End
```

但是对大多数程序来讲，只用顺序结构是不够的。因为在许多情况下，程序执行的顺序要依赖于输入数据，或中间运算结果。这要根据某个变量或表达式的值作出判定，决定执行某些语句，或跳过哪些语句。这就是程序的选择结构。

### § 3.1 程序的选择结构

所有的 Turbo Pascal 控制结构，除了 Goto 语句外，都要根据条件语句。条件语句也称布尔语句，是其结果为 True 或 False 的表达式。布尔表达式都是用逻辑运算符连接的。布尔表达式可以由变量，常数或数值直接比较组成。如：

age>20

i<j

Name='Jones'

其中也可以包括计算。如：

x>(y\*13)

(x-15)<>(y\*20)+sqr(z)也可以有多重条件。如

(age>197) And (Name='Jones')

逻辑运算在上一章中已作了介绍。参加运算的两个表达式要求是相容的。实型与串，串与整型是不能比较的。但实型，整型和字节可以出现在一个表达式中，因为它们是相容的。简单的布尔表达式可以只有一个操作数，如(i>0)。这种表达式很清楚，也很好理解。但对于复杂

的表达式,如果处理不当,会产生不希望得到的结果,因为这里与各种运算的优先级有关。例如

```
Var
    i,j:Integer;
Begin
    Clrscr;
    i:=9;
    j:=-47;
    WriteLn('i or j>0=', i or j>0);
    WriteLn;
    WriteLn('(i>0)or(j>0)=', (i>0)or(j>0));
end
```

在这两个布尔表达式中,都是要测试  $i$  或  $j$  是否大于 0。由于  $i$  已经赋予大于 0 的值。可能认为两个表达式的值都为真。但是第一个表达式的结果为 False。因为算术运算的优先级高于逻辑运算。而算符 And 和 or 即是算术运算符,也可以是逻辑运算符。这依据如何使用它们而定。在第一个例子中,算符 or 出现在两个整数之间,因此 Turbo Pascal 把它作为算术 or 对待。当  $i$  等于 9,  $j$  等于 -47 时,  $i$  or  $j$  的算术 or 的结果是 -39。而 -39 小于 0, 所以第一个布尔表达式的结果为假。

在第二个表达式中,  $i$  和  $j$  的测试分别放在两个布尔表达式中,并用括号括起来。在这种情况下,算符 or 作为逻辑运算对待,1 和 0 比较,结果为真,然后和  $i$  和 0 比较,结果为假。两个结果再做逻辑 or 运算,最后结果为真。

Turbo Pascal 中运算的优先级是很重要的,在一个复杂表达式中,一定要注意运算的次序,否则会产生意想不到的错误结果。在 Turbo Pascal 中,各种运算的优先级如下:

第 1 级	( )
第 2 级	函数
第 3 级	Not
第 4 级	And, *, /, div, Mod
第 5 级	or, +, -
第 6 级	<, <=, =, >, >=, <>

Turbo Pascal 中,首先计算括号 ( ) 中的表达式。因此,可以用括号改变运算的次序。同时,使用括号,可以使表达式更清晰易读。在同级运算中,次序是先左后右。

运算符 Not 使布尔表达式的值变后。如果表达式的值为真,Not 把它变为假。如果结果为假,Not 把它变为真。如  $i:=9$ 。则  $(i>0)$  为真,而  $\text{Not}(i>0)$  为假。

运算符 Not 虽然很有用,但却不是必须的,对每个使用 not 的表达式,都有一个与之等价的不使用 Not 的表达式。如:  $\text{Not}(i>0)$  与  $i\leq 0$  等价。使用 Not 可以增加布尔表达式的可读性,但是也不必要地增加了复杂性。

有些程序中所需要的布尔表达式可能很复杂,或者在程序中多处地方需要相同的布尔表达式。这时可以使用布尔函数。布尔函数可以减少出错的可能性,修改程序也比较容易。下面是使用布尔函数的例子:

Program Example;

```

Uses CRT;
Var
    CarsperHour
    Population density,
    TaxRate,
    LandCostperSquare—Foot,
    LaborcostperHour; Real;
Function qualifies; Boolean;
Begin
    Qualifies := (Cars—perHour > 1000) And
        (Population Density > 500) And
        (TaxRate < 0.10) And
        (Landcostpers—quare—Foot < 150) And
        (LaborcostperHour < 6.50)
End;
Begin
    Clrscr;
    WriteLn('Enter number of cars per hour:');
    ReadLn(Cars PerHour);
    Writeln('Enter POpulation density per square mile:');
    ReadLn(Population Dencity);
    Writeln('Enter Tax Rate:');
    ReadLn(TAXRate);
    WriteLn('Enter Laborcost—per hour:'); ReadLn(LaborcostperHour);
    WriteLn;
    If Qualifies Then
        WriteLn('Good site! ')
    Else
        WriteLn('Forget it');
    WriteLn;
    Writeln('press Enter...');
    ReadLn;
End

```

在选择程序结构中最常用的语句是 If 语句，If 语句有两种形式。其中较简单的形式为：

If <条件> Then <语句>

条件为一个布尔表达式。Turbo Pascal 首先计算这个布尔表达式，如果结果为真，则执行语句；如果结果为假，则跳过语句不执行，而执行 If 语句后面的语句。这里 Then 后面的语句可以用 Begin 和 End 括起来的程序块。

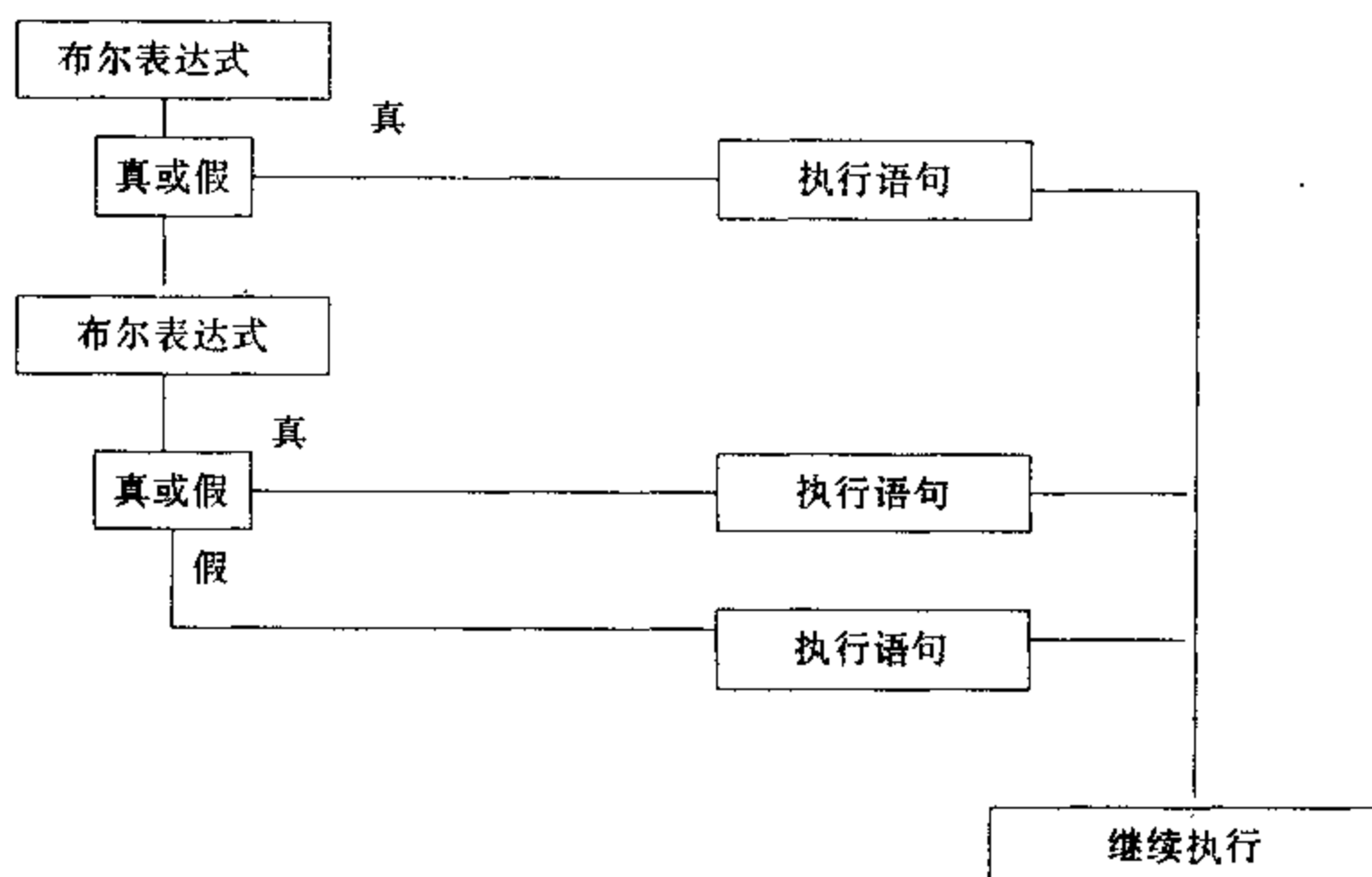
这种 If—Then 语句只有一个分枝，当布尔表达式为真时执行这个分枝。在许多情况下，程序需要两个分枝，当布尔表达式为真时执行一个分枝，当结果为假时，执行另一个分枝。这

时可用 If 语句的另外一种形式:

```
If<条件>  
  Then<语句 1>  
  Else <语句 2>
```

当条件成立时, 执行语句 1, 然后跳过语句 2, 执行其后面的语句。当条件不成立时, 跳过语句 1, 执行语句 2, 然后执行后面的语句。同样, 这里的语句 1 和语句 2 可以用 Begin 和 End 括起来的程序块。这里需要注意的是, Else 前的语句不用分号结束, 因为 Turbo Pascal 认为 If-Then-Else 结构是一个连续的语句, Else 是这条语句的一部分, 分号只出现在一条语句的结尾处。

If-Then-Else 语句提供了两个分枝, 程序有时可能需要一连串分枝。这时可以扩展 If-then-Else, 使用 Else 语句。Else 语句可以接布尔表达式, 这样程序可以有多种分枝。这种由 Els-Ife-If 语句构成的多重分枝执行的顺序加下图所示:



下面的程序表明怎样用 Else-If 建立多重分枝的条件语句。

```
Program A  
Uses CRT;  
Var  
  Grade : Char;  
Begin  
  Clrscr,  
  Write('Enter your Grade:');  
  ReadLn(Grade);  
  Grade := Uppcase(Grade);  
  If Grade = 'A' Then  
    WriteLn('Excellet.')
```

```

Else If Grade = 'B' Then
    WriteLn('Getting there. ')
ElseIf Grade = 'C' Then
    WriteLn('Not too bad. ')
Else If Grade = 'D' Then
    WriteLn('Jast made it. ')
Else If Grade = 'F' Then
    WriteLn('Summer school! ')
Else
    WriteLn('That')not a Grade. ');
WriteLn;
WriteLn('Press ENTEN... ');
ReadLn;
End.

```

If-Then-Else 语句非常有用，可以清楚地在使用大量判断。在上面的程序中，共用了三个布尔表达式，产生了六个分枝。

对于更复杂的分枝，可以用嵌套 If-Then 语句实现。嵌套 If-Then 的形式是：

```

If <条件 1>
    Then If<条件 2>
        Then <语句 1>
    Else <语句 2>
Else If<条件 3>
    Then <语句 3>
    Else<语句 4>

```

正确地使用嵌套的 If-Then 语句，在一些情况下可以减少程序的工作量，节省程序的执行时间。例如下面的例子。某场比赛分两个阶段进行，第一个阶段的前两名进入第二阶段比赛，并将第一阶段的名次化作积分带入第二阶段。第一名得 3 分，第二名得 1 分。在第二阶段比赛时，第一名得 7 分第二名得 5 分。最后计算前二名的总得分。这个程序可以如下写

```

If(Point1=1)And(Point2=1)Then
    Begin
        Point:=3+7;
        WriteLn('Point=', Point);
    End
Else If(point1=1)And (Point2=2) Then
    Begin
        Point:=3+5;
        Write('Point=', Point);
    End;
Else If (Point1=2) And (Point2=1) Then Begin
    Point:=1+7;

```

```

    WriteLn('Point='Point);
End
Else If (Point1=2) And (Point2=2) Then
    Begin
        Pion:=1+5;
        WriteLn('Point=', Point);
    End.

```

这个程序结构清楚，能很好地完成工作。但是使用嵌套更有效。如

```

If(Point1=1) Then
    Begin
        If (Pint2=1) Then
            Begin
                Poin:=3+7;
                WriteLn('Pon:= 'Point);
            End
        Else
            Begin
                Point:=3+5;
                WRiteLn('Point=', Point);
            End
        End
    Else
        Begin
            If (poin2=1) Then
                Begin
                    Point:=1+7;
                    WriteLn('Point=', Point);
                End
            End
        End
    Else
        Begin
            Point:=1+5;
            WRiteLn('Point=', Pint);
        End
    End.

```

第一个程序到每个分枝要计算三个布尔表达式，这个程序只需计算二个，在这种嵌套的 If-Then-Else 语句有可能产生错误。因为 If 语句有两种形式，一种为 If-Then，另一种为 If-Then-else。下面的格式可能产生不同的理解。

```

If<条件1>
    Then If <条件2>
        Then <语句1>

```

Else <语句2>

这时,最后的一个 Else 是与第一个 Then 配合,还是与第二个 Then 配合。这两种理解执行的效果是不同的,如果与第一个 Then 配合,条件1 为假时执行语句2。如果与第二个 Then 配合,条件1 为假时什么也不执行。

为了避免这种二义性,Turbo Pascal 规定:Else 总是与同一层前面最靠近它的,还没有 Else 与之配合的 Then 配合。因此在上面的形式下,Else 总是与第二个 Then 配合,如果要与第一个 Then 配合应使用 Begin-End。如

```
If <条件1> Then
  Begin
    if <条件2> Then
      <语句1>
    End
  Else <语句2>
```

实现选择结构程序设计的另一种方法是使用 Case 语句。这种语句有时比 If 语句更简单直观。Case 语句提供了一个多重分枝的清晰的逻辑结构。Case 语句的一般形式是

```
Case <表达式> of
  <值表1>: <语句1>;
  ... ..
  <值表 n>: <语句 n>
Else <语句 n+1>;
End.
```

Case 语句头中的表达式必须是有序类型,如整型,字符型等等。值表是一些由逗号分开的常数。这些常数为表达式可能出现的值。每个值只能出现一次。当表达的值某个值表中的值时,执行相应的语句,然后执行 Case 语句后面的语句。当有 Else 语句,并且表达式的值不在任何一个值表中时,执行 Else 后的语句。

Case 语句的一个最大的特点是,它具有说明范围的能力。如:

```
Case Key of
  'A'..'Z';
    WriteLn('You pressed an uppercase Letter');
  'a'..'z';
    WriteLn('You pressed a lowercase letter');
  '0'..'9';
    WriteLn('You pressed a numeric Key');
Else
  WriteLn('You Pressed an unKnown Key');
End
```

字符型变量 Key 中存贮一个用户键入的字符。Case 语句按照定义的范围区分字符。

## § 3.2 程序的循环结构

循环结构也是程序的基本结构。所有的循环语句如果用顺序程序结构写，不但费时费力，而且对于大量计算处理都写成一条一条的语句并输入计算机有时简直是不可想象的。

利用循环结构程序设计，可能只需写少量语句，使计算机重复执行它多次，完成大量的计算工作。在 Turbo Pascal 中，实现循环主要使用三种语句：For—Do, repeat—until 和 while—do。

### § 3.2.1 For—Do 循环

For—Do 循环是 Turbo Pascal 中最常用的循环结构。它的一般形式为：

```
For<循环变量>:=<初值>To<终值>Do  
    <循环体>
```

对 For—Do 循环，首先将初值赋给循环变量，然后将循环变量与终值比较，当循环变量小于等于终值时，执行循环体。每次执行循环体后，将循环变量的后继值赋给循环变量，再与终值比较。直到循环变量大于终值，结束 For—Do 循环，继续执行其后面的语句。

循环变量与初值、终值的类型必须相同，且只能为有序类型。循环变量的初值和终值可以是表达式。在执行循环之前，根据初、终值表达式计算循环的初值和每值，在循环体中对初、终值表达式值的改变不影响循环次数了循环变量的取值。在循环体中，不允许任何语句改变循环变量的值。在这种格式的 For—Do 循环中，如果初值大于终值，循环体将不执行。

除了上面的形式外，For—Do 循环还有另外一种形式：

```
For<循环变量>:=<初值>Down To<终值>Do  
    <循环体>
```

这种循环是从较大的初值开始，递减到终值。与前一种循环相同，循环变量的开始值为初值，然后将循环变量与终值比较。如果大于等于终值，则执行循环体，否则结束循环。下面是一个循环结构程序的例子：

```
For I:= 1 to 10 Do
```

```
    Begin
```

```
        ReadLn(x);
```

```
        SUM:=SUM+X;
```

```
    End
```

```
    WriteLn('SUM='SUM);
```

这个程序接受输入的十个数，然后求这十个数的和，并将结果输出。

### § 3.2.2 Repeat—Until 循环

For—Do 循环使程序的功能明显地得到了改进，但是它也有一定的限制。它的循环次数由初值和终值决定。在循环中不能根据情况改变。使用 Repeat—Until 循环可以减少这种限制。Repeat—Until 循环的一般形式为：

```
Repeat
```

```
    <循环体>
```



Until <布尔表达式>

循环从 Repeat 开始, 执行到 Until 后, 检查 Until 语句中的布尔表达式。如果为假, 返回到 Repeat 指令继续执行循环体。如果为真, 执行后面的语句。在循环体中应该有改变布尔表达式值的语句, 否则程序可能进入死循环。

Repeat—Until 循环的主要优点在于不必事先指定循环的次数。它重复到布尔表达式为真时为止。如上节的例子中必须输入十个数。在那个程序中若使用 Repeat—Until 语句, 可以计算任意个数字的和, 直到输入一个0为止。

```
SUM:=0;  
Repeat  
  Readln(x);  
  Sum:=sum+x;  
Until x=0;  
Writeln('SUM=', Sum);
```

### § 3. 2. 3 While—Do 循环

While—Do 循环与 Repeat—Until 循环相似。都可以在不知道循环次数的情况下执行循环。不过这两种循环有两点不同; 首先, While—Do 循环在执行循环体之前检查布尔表达式。而 Repeat—Until 循环是在执行循环体之后检查布尔表达式。因此, While—do 循环可以一次也不执行, 而 Repeat—Until 循环至少要执行一次。其次, While—do 是在布尔表达式为真时执行循环体。为假时结束循环。Repeat—Until 在布尔表达式为假时执行循环体, 为真时结束循环体。While—Do 循环的一般形式为:

```
While<布尔表达式>Do  
  <循环体>
```

将前面的例子用 While—Do 循环重写一次。

```
sum:=0;  
i:=1;  
ReadLn(X);  
While x<>0 Do  
  Begin  
    Sum:=sum+x;  
    i=i+1  
  ReadLn(x);  
End;
```

循环语句是可以嵌套的, 即在一个循环体内还有一个循环体。这种结构也称多重循环。实现多重循环时, 可以使用前面的三种循环。在实现多重循环时要注意内、外循环的关系, 不要搞错。多重循环的嵌套次数是任意的。下面是一个多重循环的例子。

```
Program Godbah  
Var  
  N,P,Q,J:Integer;  
  FlagQ, Flagp:Boolean;
```

```

Begin
Readln(N);
P:=1;
Repeat
    P:=P+1;
    Q=N-P;
    Flagp:=True;
    For J:=2 To Round (sqrt(p)) Do
        If P Mod J=0 Then
            flagp:=False;
            FlagQ:=True;
    For J:=2 To round(sqrt(R)) Do
        If Q Mod J=0 Then
            FlagQ:=False;
Until flagP and FlogQ;
WriteLn(N, '=', p, '+', Q);
End.

```

这个程序是证实哥德巴赫独想。首先输入一个大于3 的偶数，然后寻找两个质数，使它们的和等于输入的偶数。在这个程序中，外层循环用 Repeat—Until。内层有两个并列的循环。都使用 For—Do 循环。

### § 3.3 非结构分枝

所谓非结构分枝是指程序从一点直接跳到另外一点。这个过程也称直接转移或无条件分枝。后一种讲法容易引起误解，因为非结构分枝经常与布尔表达式一起使用。通过 Goto 语句与标号，Turbo Pascal 允许有非结构分枝，标号用于指明程序将转移到的地方。在 Turbo Pascal 中，标号用 Label 说明，标号之间用逗号分开，最后用分号结束。其一般有式为：

```

Label
    <标号1>, <标号2>, ... <标号 n>;

```

在程序希望转移到的地方可以使用标号，这时只要在所要转移到的地方加上标号，后面跟冒号即可。在程序的其它地方，只需使用 Goto<标号> 语句即可转移控制到所需之处。例如：

```

i:=1;
WriteLn(i);
Point1:
B:=j/3;
... ..
Goto Point1;
... ..

```

在执行到 Goto Point1 语句时，程序将转移到 Point1: 处继续执行。

Turbo Pascal 是一种结构化的语言,不使用 Goto 语句照样可以编写程序。但是在许多情况下,使用 Goto 语句比使用其它控制结构更灵活,写出的程序更精巧。但是对使用 Goto 语句, Turbo Pascal 作了限制,即不允许跳到当前程序块外, Turbo Pascal 允许跳到同一程序或过程中的化一点,但不能从一个过程跳到另一个过程。这是为了防止乱用 Goto 语句。下面是使用 goto 语句和不使用 Goto 语句的例子:

使用 Goto 的例子:

```
For i:=1 To 25 do
  For j:=1 To 25 Do
    For K:=1 To 25 Do
      If m[i,j,k]=1 Then
        Goto Found;
Found;
  WriteLn('Found at', ' ', i, ' ', j, ' ', k');
```

不使用 Goto 的例子:

```
i:=1;
j:=1;
k:=1;
While(i<=25) And (m[i,j,k]<>1) Do
  Begin
    While(j<=25) And (m[i,j,k]<>1) Do
      Begin
        While(k<=25) And (m[i,j,k]<>1) Do
          k:=k+1;
        If(m[i,j,k]<>1) Then
          Begin
            k:=1;
            j:=j+1;
          End;
        End;
      If(m[i,j,k]<>1) Then
        Begin
          j:=1;
          i:=i+1;
        End;
      End;
    WriteLn('found at:', i, ' ', j, ' ', k);
```

这两个程序都是在一个三维数组中寻找一个值为 1 的元素。使用 Goto 语句的一个原则是程序的跳转点应是离 Goto 语句近的分枝。

使用 Turbo Pascal 标准函数 Exit 可以从当前块中退出,因此 Exit 语句也可的说是一个无条件分枝语句。下面是一个使用 Exit 函数的例子。

```
For i:=1 To 100 do
  Begin
    If(Numbers2[j]=0 Then
      Begin
        WriteLn('error;Division by zero');
        Exit;
      End;
    Numbers3[j]:=Numbers1[i]Div Numbers2[i];
  End;
```

Turbo Pascal 提供了结构化和非结构化两种方法，使用它们可以编写清晰简法的应用程序。

## 第四章 Turbo Pascal 6的集成开发环境

在 Turbo Pascal 6 中,集成开发环境作了全面修改。最明显的改变是使用了多重并行窗口,可以同时看到几个源文件。或在不同的窗口观察用一个源文件的不同部分,因而使修改程序更容易。增加的另一功能是从一个窗口向另一个窗口拷贝文本。

Turbo Pascal 6的另一个特点是支持鼠标器。对 Turbo Pascal 6来说,虽然鼠标器不是必须的,但是有了鼠标器使用集成开发环境要方便得多。

Turbo Pascal 6 使用下拉菜单系统,支持热键功能。主菜单包括如下选择:File, Edit, Search, Compile, Debug, Options, Window 和 Help。对于使用过 Turbo Pascal 以前版本的人,对其中的大部分是熟悉的。但是增加和加强的地方也很多。例如 Edit 选择中增加了窗口之间拷贝,切割和粘贴文本的选择项。此外,还有一个全新的选择项 Window,用于管理 Turbo Pascal 6 的多窗口编辑环境。

F10键激活主菜单,当前选择项用高亮度显示。用“←”和“→”键可以移动高亮度光标,用 ENTER 键选择。另一种更直接的选择方法是:在按 F10键后接着按所需选择项的第一个字母,如要选 File,按 F10后按 F,等等。

在集成开发环境下,屏幕的最低行显示根据当前环境选择的最常用的热键,列出的热键根据当前活动窗口改变。例如,Watch 窗口提示的热键与文本编辑窗口提示的不同。在任何情况下,都可以通过相应的键选择提示的热键功能。

### § 4.1 file 菜单

进入 File 菜单的热键是 ALT-F。这个菜单包括打开源代码文件将源文件记盘的功能。此外,在这个菜单中还可以改变当前目录,打印文件,得到正在工作的文件的信息,临时退到 Dos,或完全退出 Turbo Pascal 6。

File 菜单是下拉式菜单,选择项垂直排列。选择选择项时,可以用‘↑’或‘↓’键使相应选择项为高亮度,然后按 ENTER。某些选择项有相应的热键。

Open... F3

Open... 选择项可以从列表中选出一个文件,并在一个窗口中将它打开。在选择了 Open... 后,将打开一个对话框。在这个盒子顶部是一个名字区,在这个区可以指定文件名或带通配符 \* 和? 的文件标识符。在名字区的下面是所有与文件标识符匹配的文件的列表。用 TAB 键和箭头键可以在文件列表上移动光标并选择所需文件。如果匹配的文件显示不下,可以用文件名下面的滚动条使窗口滚动。滚动条两端是箭头,箭头之间是选择区。

在打开的文件对话框右侧是四个开关键。—Open, Replace, Cancel 和 Help。在集成开发环境中,很多窗口都有同样的开关键。可以通过按所示的字母(如对 Open 按 O 键),或者用 TAB 键到所需开关并按 ENTER 键,选择所需开关。Open 开关键将产生一个新的窗口,装入所选文件。Replace 键将把选定的文件装入当前的编辑窗口,如果有的话,Cancel 键关闭窗口,

不选文件。Help 将提供与当前窗口有关的帮助信息。

#### New

当选择 File 菜单中的 New 后，将打开一个空的文本窗口，这个文本文件带缺省文件名 NONAME.PAS。最多可以打开一百个这样临时编译窗口，它们将顺序编号到 NONAME99.PAS。当把一个新窗口的内容记盘时，要重新对文件命名。

#### Save F2

Save 选择项把当前编辑窗口的内容记到磁盘上。如果当前编辑窗口是用 New 选择项创建的，则要求输入新的文件名。

#### Save as...

与 Save 选择项一样，Save as... 选择项也是把当前编辑窗口的内容记到磁盘上。但是要求输入文件名。名字可以是新的，也可以是已经存在文件的名字。一旦选定一个文件名后，所有编辑窗口对那个文件所进行的修改都将反映选出的新名。

#### Save all

Save all 选择项将把所有正在编辑的文件内容记盘，只要文件以某种方式进行了修改。

#### Change dir...

Change dir... 选择项用一个改变目录对话框改变当前目录。可以输入所需路径改变目录，也可以由对话框显示的目录树改变目录。当选择一个目录后，也可以用 Revert 开关键取消选择。

#### Print

Print 选择项把当前编辑窗口的全部内容送到打字机。但这个命令不能输出打印窗口或帮助窗口。

#### Get Info...

在编译程序时，Get Info... 选择项提供有关正在编译的程序的信息，包括编译的行数，代码数据，现栈所占的内存量，及堆所需的最大量和最小量。这个窗口还显示系统内存与扩展内存的使用情况。

#### Dos shell

Dos shell 选择项可以暂时将 Turbo Pascal 挂起来，进入 Dos 提示符，这时可以使用 Dos 命令或运行其它程序，在 DOS 提示符下输入 EXIT 可返回集成开发环境。

#### Exit Alt-X

Exit 选择项的热键是 ALT-X。它将终止 Turbo Pascal 的集成开发环境，把控制返回到 DOS。当选择 Exit 时，如果某个窗口含有修改过的源文件，将提示是否希望在退出之前记盘。

## § 4.2 Edit 菜单

进入 Edit 菜单可以用热键 ALT-E。Turbo Pascal 6 允许同时在不同的窗口打开多个源文件。在 Edit 菜单下的一个特殊的缓冲区，叫作记录板。使用记录板可以从一个编辑窗口向另一个窗口传送文本块。在从一个窗口向另一个窗口拷贝块时，首先选出文本块。然后把这块文本拷贝到记录板中。接着，在另一个源文件中找到需要拷贝的位置，用 Paste 选择项把文本块从记录板中拷贝出来。

选择文本块有几种方法。首先把光标放在这个块的开始处。选择一个字时按 CTRL-KT，

选择一行按 CTRL-KL。选择更大的块时按 CTRL-KB，然后把光标移到这个块的结束处按 CTRL-KK。

#### Restore Line

Restore Line 选择项用于取消编辑窗口中一行所作的改变。这只对最近的文本编辑行有效。

#### Cut SHIFT-DEL

Cut 选择项把当前编辑窗口中选择的文本块移到记录板中。移出的文本块可以移到编辑文件的其它地方，或移到另一个窗口的文件中。

#### Copy CTRL-INS

Copy 选择项与 Cut 选择项一样，但是它不把选出的块从源文件中去掉，只是把这个块的付本放入记录板。这个块可以从记录板中粘贴到文件的其它地方，或另一个窗口的文件中。

#### Paste SHIFT-INS

Paste 选择项从记录板中向当前编辑窗口拷贝文本，拷贝的文本从窗口的光标处开始。文本粘贴到窗口之前，首先必须从一个现在文本文件中拷贝或切割下来。Paste 选择项只使用最后加到记录板中的文块。

#### Copy example

Turbo Pascal 6 的帮助系统除了提供信息外，还提供例子，演示如何使用函数和过程，使用 Copy example 选择项，可以把举例代码拷贝到记录板，这样就可以把它粘贴到程序中，举例代码是事先选定的，在拷贝之前不必再选择。

#### Show clipboard

记录板缓冲区已有已经从文本文件上拷贝或切割的所有文本。它实际上是一个特殊的编辑窗口。Show Clipboard 选择项打开这个窗口，看到它的内容。某些文本是高亮度时，这是等待粘贴到编辑窗口的文本。如果需要可以修改这一文本。

#### Clear CTRL-DEL

Clear 选择项从当前编辑窗口去掉选出的文本，但不拷贝到记录板。结果删除的文本不能恢复。

## § 4.3 Search 菜单

进入 search 菜单的热键是 ALT-S，开发程序的大量工作是找出错误并确定它的位置。search 菜单提供一套工具帮助完成这些工作。

### § 4.3.1 find... CTRL-QE

Find... 选择项用于在源文件中为一个文本模式定位。这个选择项打开一个搜索对话框，指出要搜索什么及如何完成搜索。在对话框顶部有一个区域，用于输入想要搜索到的文本。下面是设置控制搜索的选择项。

#### Options

下面是控制搜索的选择项：

Case Sensitive 如果希望区分大小写字母，关闭这一选择项。

Whole Words Only 如果关闭这一选择项，Turbo Pascal 将只匹配由空格和标点符号括起

来的那些串。

Regular expression 关闭这项功能,将使用扩展表达式匹配,可以使用下表中的特殊字符。

表:正规表达式搜索的通配符

通配符	说明
^ (脱字符)	只匹配一行的开头。如 ^ ABC 正配 ABCX, ABCYY 等等。
\$ (币符)	只匹配一行的结尾。如 ABC \$ 匹配 ABC, XABC, YYABC 等。
.(句号)	匹配任何字符。如 AB. 匹配 ABA, ABB, ABC 等等。
* (星号)	匹配指定字符的任意个数出现包括不出现。如 ABC * 匹配 AB, ABC, ABCD, ABD 等等。
[] (方括号)	匹配包括在括号中的任何字符。如 AB[CD] 匹配 ABC 或 ABD。
[^ ] (括号中的脱字符)	匹配不包括脱字符后的任意一字符的串。如 AB[^ CD] 匹配 ABX, ABF 等等。
[ - ] (括号中的减号)	匹配用减号指明的字符范围。如 AB[C - G] 配 ABC, ABD, ..., ABG。
(反斜线)	指出下一个字符是串的一部分,不是特殊字符。如 AB. 匹配 AB ^。

#### Scope

搜索范围可以是全局的(包括整个文件)也可以只限于文本的现行选择区。

#### Direction

搜索可以从当前光标位向后(向文件头)进行,也可以向前(向文件尾)进行。

#### Origin

搜索可以从当前光标位开始,也可以包括整个文件。

### § 4.3.2 Replace... CTRL-QA

Replace... 选择项与 Find... 一样,但是可以用新的文本替换找到的文本。replace... 选择项也打开一个替换对话框。这个窗口,几乎与搜索对话框完全相同,只是增加了两点。在搜索文本区下面还有一个区,用于键入替换旧文本的新文本。此外还有一个 Prompton replace 选择项,如果作了选择,在每次替换这前要求证实。

### § 4.3.3 search again CTRL-L

这个选择项简单地重复最后的搜索操作,可以从菜单选择这个选择项,也可以按 CTRL-L。

### § 4.3.4 Goto Line number...

Go To Line number... 选择项把光标移到所输行号的开头。



### § 4.3.5 Find procedure...

在编程时,能够快速找到过程或函数调用的位置很有用,Find Procedure... 选择项可以做到这点,只要输入所需的函数或过程的名字即可。不过这个选择项只在调试期间有效。

### § 4.3.6 Find error... ALT-F8

在出现运行错误时,错误的定位信息以段偏移量的方式显示,如 0F23:1029。要确定错误在原代码中的位置可以使用 Find error..., 或者用热键 ALT-F8。这时只要输入段和偏移量即可。应注意,只有当调试信息选择 { \$D+ } 有效时,这一选择项才能工作。

## 4.4 Run 菜单

进入 Run 菜单的热键是 ALT-R。程序写好后即可以由 Run 菜单编译运行。在调试程序时,也可以使用这个菜单。

Run CTRL-F9

Run 选择项用一条命令编译并运行程序。如果当前窗口中的文件与其它文件有关,而那些文件在最后一次编译后做了修改,则这些文件也将进行编译。如果是调试程序,Run 命令将执行程序到一个断点或到程序结束。

Program reset CTRL-F2

在调试时,可能需要从头开始执行。选择 Program reset 选择项可以做到这点

Goto Cursor F4

Goto Cursor 选择项在调试时使用。把光标置于所需的行上,按 F4,如因需要,编译这个程序,程序将执行到指定的行。

Trace into F7

Trace into 选择项执行一行程序代码。如果当前行是一个函数或过程,按 F7 将进入函数或过程

Step Over F8

Step Over 的工作方式与 Trace into 一样,但是在调用函数或过程时,它不是进入函数和过程,而是象一条表达式那样执行。

Parameters...

程序常常需要依赖命令行参数,即这些参数是在由 DOS 提示符执行程序时传递给程序的。parameters... 选择项可以在集成开发环境下指出传递给程序的参数。

## 4.5 Compile 菜单

进入 compile 菜单的热键是 ALT-C。这个菜单可以编译程序的一部分或全部,并指出被编译的程序应存贮在什么地方。

Compile 选择项将编译当前编辑窗口中的代码,其它代码不能由这个选择项编译。按 ALT-F9 可直接进入这一选择项。

Make F9

Make 选择项比 compile 选择项有效得多,它不仅编译当前窗口中的代码,而且这些代码所依赖的任何源文件,只要自最后一次编译整个程序之后作过修改,都将进行编译。Make 处理的结果是可执行程序,可以运行和调试。

#### Build

Build 选择项与 Make 一样,但是它将重新编译程序中的所有源文件,不管它是否作了修改。在改变了全局编译指令后,通常应使用 Build 选择项。

#### Destination Memory

使用 Destination Memory 选择项可以指出编译的可执行程序是存贮在内存中,还是存贮到磁盘上。如果从 DOS 提示符运行程序,必须把程序存贮到磁盘上。存贮到内存的可执行程序只能在集成开发环境中运行。

#### primary file...

如果程序由多个源文件组成,应用 Primary file... 选择项指出主要的源文件。指出主文件可以保证在发出 Make 或 build 命令时,所有的单元会适当地编译。

## § 4.6 Debug 菜单

集成调试程序是一个极为有力的工具。用它开发可靠的程序比用其它编译工具容易得多。按热键 ALT-D 可以直接进入 Debug 菜单。

#### Evaluate / modify... CTRL-F4

在调试过程中,经常需要改变一个变量的值,观察不同的值对执行程序的影响。使用 Evaluate / modify... 选择项可以观察修改一个变量的值。求值与修改对话框可以输入一个表达式进行求值。表达式可以是一个变量名或者一个完整的表达式。表达式求值后,把它的当前值显示在结果区(result)。如果想修改一个正在观察的变量的当前值,用 TAB 键到新值区,打入所需的值,按 ENTER 键。这时变量的新值会出现在结果区。

#### Watches

在调试程序中最重要的是观察变量在程序执行中是如何改变的。Watches 选择项将弹出一个菜单,用于控制想要观察什么变量。要加上一个观察变量,按 CTRL-F7并输入想要观察的变量名。这时将打开一个窗口,显示选出的变量。这个菜单的其它选择项可以从观察列表中删除一个变量,编辑变量的说明,或用一条命令删除全部观察的变量。

#### Toggle breakpoint CTRL-F8

调试程序,设置断点很重要。经常要在代码的关键位置上设置断点。断点可以是无条件的,程序执行到断点处一定停止。也可以是有条件的,程序执行到断点处只有当给定条件为真时才停止。要设置无条件断点,只要把光标置于要求的代码行上,选择 Toggle breakpoint 选择项,或者按 CTRL-F8。无条件断点一经设定,程序执行到那一行时将停止。

#### Breakpoint...

条件断点可以对调试做更多的控制,可以指定停止执行程序的条件。条件断点可以是任何值为真或假的表达式。这有助于确定值在什么地方超出了希望的范围。也可以指定通过记数值,指出停止执行程序之前,断点可以触发多少次。缺省值是0次。

## § 4.7 Options 菜单

Options 菜单是所有 Turbo Pascal 菜单中最复杂的。它有各种选择项, 控制 Turbo Pascal 环境。通常只需要改变其中的几个选择项, 但是它们合在一起使 Turbo Pascal 6 成为最全面最强大的 Pascal 环境。按 ALT-0 可进入 Options 菜单。

### § 4.7.1 Compiler...

Compiler... 选择项可以完全控制 Turbo Pascal 6 如何生成程序代码。

Force far calls—与 { \$F } 编译指令相同。设置有效时, 所有的过程与函数调用都用远程调用; 设置无效时, 同一单元内的调用为近程调用。

Overlays allowed—与 { \$O } 编译指令相同。设置有效时, Turbo Pascal 将生成支持覆盖所需的代码; 设置无效时, 不生成这样的代码。

Word align data—相当于 { \$A } 编译指令。设置有效时, 为了加快运算, 所有的非字符数据都在偶地址; 设置无效时, 数据连续放入存储空间。

286 instructions—相当于 { \$G } 编译指令。设置有效时, 代码中将包括充分利用 80286 处理器的特殊指令, 这样的程序将只能在 80286 或更高的机器上运行; 设置为无效时, 所有代码都是标准 8086 处理器的。

Range checking—相当于 { \$R } 编译指令。设置有效时, 将产生检查数组和串变量下标和标量数据类型超界条件的代码; 设置无效时, 不生成这样的代码, 程序也较小。这种类型的错误很难检查出; 所以在调试完成之前应加上这条指令。

Stack checking—相当于 { \$S } 编译指令。设置有效时, 将产生代码, 以保证执行一个函数或过程之前有足够的堆栈空间放置参数和局部变量; 如果设置无效, 将不生成这样的代码, 程序也较小。

I/O checking—相当于编译指令 { \$I }。设置有效时, 将生成代码, 在每次 I/O 操作后检查 I/O 错误; 设置无效时, 将不自动检查 I/O 错误。

Strict var—strings—与 { \$V } 编译指令相同。设置有效时, Turbo Pascal 要求作为变参传递的串应与形参定义的长度匹配。这是防止向一个参数传递大于它的串。

Complete Boolean evaluation—与 { \$B } 编译指令相同。当设置为有效时, 所有的布尔表达式在继续执行之前都将全部计算。若设置为无效布尔表达式将只计算到最后结果确定已知时为此。

Extended syntax—与编译指令 { \$X } 相同。当设置为有效时, Turbo Pascal 6 承认下面扩展语法规则: 可以象过程表达式那样使用函数调用。例如: FunctionX; Integer 可以用 X(); 调用。

8087/80287—与 { \$N } 编译指令相同。当设置为有效时, 即 8087/80287 方式, 允许使用 single, double, extended, 和 comp 数据类型。若设置为无效, 这些数据类型都不能使用。

Emulation—相当于 { \$E } 编译指令。当设置为有效时, 如果有数学协处理器芯片, 程序将使用这个协处理器。否则用软件模拟。当设置为无效时, 如果已选择 8087/80287 方式, 程序要求数学处理器。

Debug information—等价于 { \$D } 编译指令。当设置为有效时, Turbo Pascal 将产生调试

信息,并存储于.TPU文件中。在集成开发环境中调试程序时,将使用这些信息。设置为开时,调试信息不会存入.TPU文件中。

Local symbol—与{\$L}编译指令相同。当设置有效时,将产生局部符号的调试信息,这些符号是在一个过程,一个函数,或一个单元实现部分内说明的。当设置为无效时,将不产生这些信息。如果调试指令设置为无效时,这条指令将忽略。

conditional defines—这里输入的符号由编译程序在计算编译指令。\$IFDEF和\$IFDEF时使用。这与在程序中使用\$DEFINE表达式是等价的。

#### § 4.7.2 Memory size...

这个选择项与{\$M}编译指令等价。在这里可以指定程序的堆栈所用的内存量(缺省值64K),以及程序堆的最大和最小量(最小为0K;最大为640K)。

#### § 4.7.3 Linker...

Linker... 选择项控制映射文件,链接缓冲区和独立调试信息的产生。

Mapfile—指定 Turbo Pascal 6 不生成映射文件,生成只有段信息的映射文件,生成有段信息及全局变量信息的映射文件,还是生成详细映射文件,包括局部符号的信息。Linker buffer—为了加快编译,可以指定 Turbo Pascal 只使用内存做链接处理的缓冲区;如果计算机的 RAM 较少,可以选择磁盘缓冲区。

Debugging—用于指出代码是在 Turbo Pascal 6 中调试,还是独立调试,即用 Turbo Debugger 调试。

Display swapping—在调试中,必须仔细注意不要使程序的屏幕输出覆盖了调试屏幕。如果选择 None, Turbo Pascal 6 不做任何工作防止干扰调试屏幕;若选择 Smart, Turbo Pascal 将自己决定使屏幕转换减到最小;如果选择 Always, Turbo Pascal 6 将采取一切措施防止干扰调试屏幕。

#### § 4.7.4 Directorices...

Turbo Pascal 需要知道在哪里寻找.TPU文件,包含文件,源文件单元和目标代码。缺省值是当前目录,但是可以指定其它目录。如果文件不在当前目录下时, Turbo Pascal 将到那些目录中寻找。

#### § 4.7.5 Environment

环境对话框用于控制 Turbo pascal6 工作的计算机设备,包括鼠标器,以及退出时应存储的信息。

Preferences—如果 PC 想上带 EGA 或 VGA 显示卡,可以使屏幕显示为43行或50行,以代替标准的25行。也可以控制调试阶段是否打开一个新的窗口。选择 New window 可以保证每次需要一个新的源文件时,打开一个新的源窗口;选择 Current Window 在需要一个新的源文件时,将替换当前窗口中的内容。

对自动存储方式做出的选择可以确定,在退出 Turbo Pascal 或在集成开发环境中运行一程序时,什么将存入磁盘。可以指定自动存储编辑文件, Turbo 环境,屏幕窗口的配置,或这三者的任意组合。

Editor—可以控制某些编辑形式，如列表，自动缩进和插入方式。

Mosue—鼠标器的右键可以编程，完成某种功能：用它取得帮助信息（即选择 Topic Arch），在文本文件中移动光标的位置，设置断点，表达式求值，把选定的变量加到观察列表中。使用鼠标器可以节省很多时间。

Startup—在 startup 选择项对话框中的选择项可以控制 Turbo Pascal 设置其工作环境方式，其中包括内存分配，视频界面优先级和扩展内存的使用等。

Colors—色彩对话框可以指定集成开发环境的前景和背景色彩。

#### § 4.7.6 save Options...

在改变了 Turbo Pascal 选择项后，可以用 save Options... 把这些改变记到指定的文件中，若未指定文件，则记到 Turbo. TP 中。

#### § 4.7.7 Retrieve options...

使用这个选择项可以把记在磁盘文件中的选择项恢复。这样，可以把不同需要的选择项记在有关文件中，需要时恢复它们。

### § 4.8 Window 菜单

重叠窗口环境是 Turbo Pascal 6 最显著的改进。Window 菜单可以控制窗口管理的各个方面。按 ALT-W 可直接进入 Window 菜单。

Size/Move CTRL-F5

在 size/move 选择项可以用箭头键在屏幕内移动当前窗口。按住 SHIFT 键，用箭头键可以扩大或缩小窗口。

Zoom F5

用 Zoom 可以使当前窗口充满整个屏幕。再次选择 zoom 可以使之恢复到原来的大小和位置。

File

同时打开多个窗口时，会出现一点麻烦。使用 Tile 命令可以缩小所有的窗口，使它们同时出现在屏幕上。按住 ALT 键再按所选窗口号可以选择窗口。

Cascade

这一选择项是另一种安排窗口的方法。它不是缩小窗口，而是把它们串起来，使每个窗口的最上一行都可见。

Next F6

按 F6 键可以快速移到下一个窗口。

Preuious SHIFT-F6

按 SHIFT 和 F6 键可以快速移到前一个窗口。

Close ALT-F3

Close 选择项关闭当前窗口。如果窗口中有修改过的文件，将提示在关闭窗口之前是否将文件记盘。

Watch

这个选择项打开或查看 Watch 窗口。

#### Register

这个选择项打开或查看 Register 窗口。这个窗口显示所有 cpu 寄存器的状态，这对寻找程序中的缺陷很有用。

#### Output

这个选择项打开一个窗口，显示“Turbo Pascal 外部”屏幕。即在装入 Turbo pascal 6 时的屏幕内容，这时任何屏幕输出都由程序产生。

#### Call Stack CTRL-F3.

只有在调试时，才有 Call Stack 窗口，它对确定程序的位置很有用。这个窗口将显示过程和函数调用的层次。

#### User Screen ALT-F5

User Screen 选择项使整个屏幕由用户程序使用，按 ESC 键回到 turbo Pascal 6 的集成环境。

#### List... ALT-0

这个选择项显示打开窗口的所有文件的列表，其中也包括现已关闭的窗口的文件。

## 第五章 指针与动态内存分配

Turbo Pascal 把计算机内存分为四个部分：代码段，数据段，堆栈段和堆。每个部分都有其专门的作用。

代码段存放程序，主程序和程序的每个单元都有自己的代码段。只有一个数据段，用于存放类型常量和全局变量。除了数据段外，其它地方也可以存放数据。堆栈和堆可放动态数据。堆栈很重要，它的操作由 Turbo Pascal 自动控制，程序不能改动。对高级编程技术来讲，堆是特别重要的。

### § 5.1 Turbo Pascal 的内存分配

计算机都有一定数量的随机存储器 RAM。在程序启动时，首先建立一个或几个代码段，存放程序；一个数据段，存放程序的数据；和一个堆栈段，存放临时数据。在执行代码段的指令的同时，处理数据段和堆栈段的数据。

确定任何一个特定字节要使用地址。每个字节都有一个地址。当程序需要存取特定字节时，用地址找到这个字节在内存中的位置。

早期的8位微处理器中，地址由一个字(两个字节)组成，只能存取64K(65536)个字节以内的 RAM。

后来的16位微处理器，例如8086/88系列，使用一种新的内存寻址方式，使用段地址。段地址把两个字的值结合成一个20位的地址，其中一个字为段，另一个字是偏移量。每个段有64K的 RAM，8086/88处理器可以有16个段，可以有1M(1,048,560)字节的 RAM。但是DOS限制使用640K以上的内存。

Turbo Pascal 提供了两个标准函数 Seg 和 Ofc，Seg 返回变量所在的段，Ofc 返回偏移量。下面的程序表明如何使用这两个函数求变量的地址。

Program Addresses;

Use CRT;

Type

StType=String[10]

Var

i;Word;

s;String[5]

r;Real;

Type

St4=String[4]

(\*\*\*\*\*)

Function intToHex(i;word);St4

```

Var
HexStr:String[8]
b:Array [1..2] of Byte Absolute i;
bt:Byte;
( * * * * * )
Function Translate(b:Byte):Char;
Begin
If b<10 then
    Translate:=chr(b+48)
Else
    Translate:=Chr(b+55);
End;
( * * * * * )
Begin
HexStr:='';
HexStr:=HexStr+Translate(b[2]Shr 4)
HexStr:=HexStr+Translate(b[2]And 15)
HexStr:=HexStr+Translate(b[1]Shr 4)
HexStr:=HexStr+Translate(b[1]And 15)
IntToHex:=HexStr;
( * * * * * )
Begin
ClrScr;
Writeln('Word:      ',IntToHex(Seg(i)),':',IntToHex(Ofs(i)));
Writeln('String:     ',IntToHex(Seg(s)),':',IntToHex(Ofs(s)));
Writeln('Real:        ',IntToHex(Seg(r)),':',IntToHex(Ofs(r)));
Writeln('Word:        ',IntToHex(Seg(c)),':',IntToHex(Ofs(c)));
Writeln;
Write('Press ENTER...');
Readln;
End.

```

这个程序定义了四个不用类型的变量。Seg(i) 求出变量 i 的段地址，Ofs(i) 返回其偏移量。函数 IntToHex 接受一个字参数，返回用这个字符的字符串表示的十六进制的值。段和偏移量通常都用十六进制格式表示。由于系统的不同配置，变量的地址会有不同。下面是运行上面程序屏幕输出的一种结果：

```

Word:          68Bb:003c
String:        68BB:003E
Real:          68Bb:0044

```

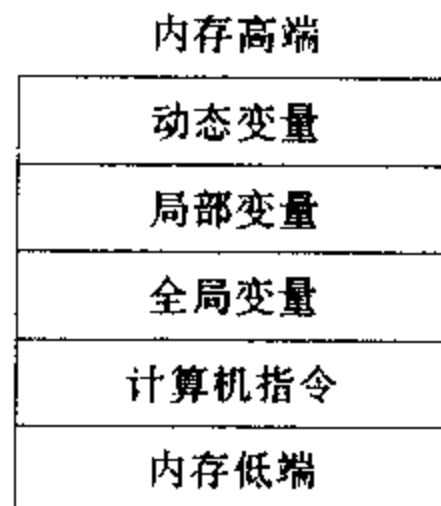


所有的全局变量都有相同的段，偏移值之间的距离与存贮每一变量类型所需的字节数相合。例如字的偏移量从3ch开始，一个字需要两个字节存贮，则下一个变量的偏移量从3Eh开始。

Turbo Pascal 中的类型常量存贮在数据段中，而无类型常量在代码段中。实际上，无类型常量是计算机代码的一部分。因此，无类型常量没有地址。

在过程和函数中说明的变量存贮在动态数据存贮区，即堆栈中。当程序调用一个过程时，Turbo Pascal 将在堆栈中为这个过程的局部变量分配空间。当堆栈中增加变量时，堆栈向下生长。过程结束后，这些变量将释放，内存可以再次使用。

Turbo Pascal 内存中的第四段是堆，这也是用户可用的动态数据区。堆有效地利用内存，因为它不必在程序中从头至尾保存数据结构。用户可以在堆中建产一个变量，并随时可以把它从堆中去掉。释放的空间可以为另一个变量使用。下面是 Turbo Pascal 的内存结构图。



代码段位于内存的低端，接下来是数据段和堆栈段。最后，堆位于内存的高端。堆的最大量由编译指令 M 指定。图中还表明，堆栈是向下生长的。而堆是向上生长的。

## § 5.2 堆和指针

Turbo Pascal 中，大部分变量都说明为静态。它们在程序执行中始终占据所分配的内存。而堆使用指针这样一种动态数据类型。在程序运行中可以建立或释放指针变量。不同的指针变量都可以使用堆中的内存。

使用指针有两个好处，首先它增加了程序可用的内存量。数据段只能使用64K字节，而堆只受计算机 RAM 总量的限制。其次使用指针可以使程序在较小的内存中运算。例如一个程序需要两个非常大的数据结构，但是一次只使用其中的一个。如果把它们说明为全局变量，就在数据段内，并始终占据内存。如果把它说明为指针，就可以把它们放在堆中。必要时可以把它们释放掉，因此减少了内存需求量。

与其它变量不同，指针变量中不是数据而是指向堆中变量的地址。定义指针的一般形式为：

<指针变量名>: <类型>

例如：

i: ^ Integer

则 i 是一个指向整型数的指针。i 本身是一个地址，而不是一个整型数。地址所指的单元才是整型数，在使用指针之前，首先应用 New 语句为指针变量在堆中分配一个存贮地址的单元。指针运算如下：

i^ := 100;

其中 ^ 表示对指针所表示的堆中变量进行操作，而不是对指针本身进行操作，而改变指针本身的运算如下：

i := 100;

这个表达式是对指针操作，它使指针指向内存中100处，而不是 i 应指的位置，下面这个例子说明了变量在堆内是如何分配和释放的。

Type

St10 = string[10];

Var

i: ^ Integer;

r: ^ real;

S10: ^ st10;

W: ^ word;

New(i) New(st10) New(r), Dispose(i) New(W)

在这个例子中，定义了四个指针变量 一个整型，一个实型，一个字符串，和一个字。当执行 New(i) 时，Turbo Pascal 在堆中分配二个字节，正好存放整型变量 i。执行 New(st10) 后，由于 stn 是有10个字符的串，它要占11个字节，其中一个字节放串长度。因此在堆中为 st10 分配11个字节。执行 New(r) 后，再为 r 分配5个字节，因为 r 是实型占5个字节。执行 Dispose(i) 后，Turbo Pascal 将释放 i 所占的空间。这样 i 原来所占的两个字节就空了出来，这时其它变量可以使用这两个字节。当然如果要使用这部分空出来的空间，新的数据结构必须与空出来的空间大小相合，否则只能分配在堆中的其它部分。这样 New(w) 执行后，E^ 正好占据了原来 i 所空出来的空间。

New 和 Dispose 是成对使用的。New 为变量分配内存，Dispose 将变量所占的内存释放掉，使用 New 和 Dispose 时必须小心。否则可能出现错误。如一个常犯的错误是：

New(i);

...

New(i);

这两条语句都在堆中分配 i，但是只有一个 i 可以使用。而且用户不但不能使用第一个 i，甚至不能消除它。因为 i 只指向第二个变量。

除了 New 和 Dispose 外，另一种动态分配内存的方法是用 Mark 和 Release。这种方法与 New - Dispose 方法不同，它不是在堆中留下空穴，它利用一个特殊的指针，把堆的尾部整个砍掉。下面的程序演示了如何使用 Mark 和 release:

Program HeapRelease;

Use CRT

Type

Atype = Array[1..100] of Char;

```

Var
    HeapTop; ^ Word;
    a1,a2,a3: ^ Atype;
Begin
    ClrScr;
    Mark(HeapTop);
    Writeln('Initial free memory:',MemAvail);
    Writeln;
    Writeln('-----');
    Writeln;
    New(a1);
    Writeln('Free memory after allocating a1:',MemAvail);
    New(a2);
    Writeln('Free memory after allocating a2:',MemAvail);
    New(a3);
    Writeln('Free memory after allocating a3:',MemAvail);
    Writeln;
    Writeln('-----');
    Writeln;
    Release(HeapTop);
    Writeln('Free memory after release:',MemAvail);
    Writeln;
    Write(
        'Press ENTER...');
    Readln;
End.

```

在这个程序中共有三个指针变量。使用标准函数 Memavail 显示剩下的可用内存，返回的值是用字节表示的内存量。程序开始用指针变量 HeapTop 保存从哪点开始释放内存。语句 Mark(HeapTop) 把堆顶的当前地址存贮在指针 HeapTop 中。在堆中分配任何变量之前，程序调用 Mark(HeapTop)。结果，在调用 release(HeapTop) 时，堆中所有的变量都将释放，可做它用。

运行上述程序，可产生结果如下：Initial free memory: 195632

```

.....
Free memory after allocating a1: 195532
Free memory after allocating a2: 195432
Free memory after allocation a3: 195332
.....
Free memory after realease : 195632

```

每增加一个变量，可用的内存量就相应地减少。在释放掉所有变量后，可用内存以恢复到原来的大小。

除了在程序头使用 HeadTop 指针外,也可以在分配了一些变量后再用 HeadTop 指针,这时,在 Headtop 指针之前分配的变量所用的内存不能释放。如在上例中,若在 a1 之后使用 Heaptop 指针,程序写处使用 Release 时,只能释放的和 a3, a1 将不能释放。

应该注意的是, Disptose 和 releasc 是不兼容的。在一个程序中只能选择使用其中的一种,否则可能产生内存冲突。

动态内存分配的第三种方法是使用 GetMem 和 FreeMem,这种方法也是每次分配或释放一个变量。但是, GetMem 和 FreeMem 可以指定一个特定的值,用于指定所用变量类型需要分配多少内存,下面的例子说明发中何用 GetMem 分配内存,用 FreeMem 释放内存:

```
getMem(i,20);  
i:=x+y;  
WiteLn(i);  
FreeMem(i,20);
```

语句 GetMem(i,20)给变量 i 分配20个字节。FreeMem 中指定的字节数必须与 GetMem 中指定的字节数相同,也不要 Dispose 代替 FreeMem 释放变量。否则堆中会出现混乱。

### § 5.3 链表

堆一般用来管理较大的数据空间,所以指针变量用于大而复杂的数据结构。动态数据管理的一种方法是使用链表。链表是一个序列,其中的每个元素都有两个部分,一部分是数据部分,另一部分是指针。单向链表的指针指向下一项;双盘链表中一个指针指向上一项,一个指向下一项。

使用链表有很多优点。首先可以加快程序运算的速度。最大限度地利用内存。其次在链表中插入或删除一个元素很方便。用链表排序也很容易,只需修改指针,不必整个改动链表。很多情况下,使用数组有很多限制。特别是在元素个数不定时更是如此。如果按最大元素个数保留空间,一般会浪费很多内存。若要节省内存,又有可能因为数组太小不能使用。

一个典型的单向链表至少需要三个指针:一个指向链表的头,一个指向当前记录,另一个指向前一个记录。同时,单向链表中每个记录中必须有一个指针指向下一项。例如:

```
custptr = ^ CustRec;  
CustRec = Record  
    Name: String[20];  
    Address: String[40]; City: String[20];  
    State: String[2];  
    Next: Custptr;  
End;
```

其中 Custptr 是指向记录 CustRec 的。同时,记录 CustRec 中含有域 Next 定义为 Custptr 型。在处理表的过程中,首先要知道表在什么地方开始,到什么地方结束。还应该知道,当前处于表的什么位置,以及下一个记录的位置。下面就是这三个指针的定义。

```
Var  
    FirstCust,  
    PrevCust.
```

Currentcust; Custptr;

如果从头到尾处理整个链表, 必须知道第一个记录的位置。第一个记录的指针存在放 First—Cust 中, 在程序开始时, 首先要对 First—Cust 初始化, 将 Nil 放在 Firstcart 中;

First—Cust := Nil;

当程序发现某个记录中的指针是 Nil 时, 就知道到了链中的最后一个元素。在链表中增加一个新的记录时, 要把这个链连接到前一个链上。当这个链是第一个链时是个例外, 它没有前继链。下面的程序演示了如何利用这些指针创建一个链表:

If Firstcust = Nil Then

Begin

New(currentcust);

Firstcast := CurrentCust;

CurrentCust<sup>^</sup>. Next := Nil;

End

Else

Begin

Prevcust := Currentcust;

New(currentcust);

Prevcust<sup>^</sup>. Next := currentcust;

CurrentCust<sup>^</sup>. Nex := Nil;

End

这段程序首先检查 Firstcust 是否为 Nil。如果是, 则说明这是链表中的第一个链。这时程序建立 Currentcust 记录, 并使 Firstcust 等于 Currentcust, 设置 Currentcust 中的 Next 域为 Nil。

如果 Firstcust 不等于 Nil, 它将把指针 prevcust 设置为 Currentcust, 并建立一个新的 currentcust, 它指向一个新建立的链。在建立了新的 currentcust 后, 程序将把 prevcust 中的 Next 域设置为指向这个新建立的链。这样, 表中的元素通过指针域互相连接在一起。

下面这段程序表明应如何从头到尾处理链表:

Current cust := Firstcust;

While Currentcust <> Nil Do

Begin

<处理过程>

currentcust := CurrentCust<sup>^</sup>. Next;

End;

首先使当前记录指针指向链的头, 即

Currentcust := Firstcust。然后用 While—Do 循环处理 链表, 直到 Currentcust 为 Nil 为上。在循环体中的最后一个语句 Currentcust := Currentcust<sup>^</sup>. Next 使指针指向链中的下一个记录。

若要删除一个记录或插入一个记录也很容易。在删除一个记录时, 首先使用前面的循环过程, 使 Currentust 指向要删除的记录。然后使 prevcust 中的 Next 等于 Currentcust 中 Next 指针。再用 Dispose 删除 Currentcust 指向的记录, 即完成了删除处理。下面这段程序表明如何

删除一名。Name='Johns' 的记录

```
Currentcust := Firstcust;  
While(currentcust^.Name <> 'Johns') And (currentcust^.Next <> Nil) Do  
  Begin  
    Prevcust := Currentcust;  
    Currentcust := currentcust^.Next  
  End;  
If currentcust^.Name = 'Johns' Then  
  Begin  
    Prevcust^.Next := Currentcust^.Next;  
    Dispose(currentcust)  
  End
```

这段程序首先确定要删除记录的位置。如果符合条件的记录找到,则删除这条记录。增加一个记录也很简单。首先使 Currentcust 指向将要插到其前面的那个记录。然后建立一个新记录。使前一个记录的指针指向这个新记录,使新记录的指针等于前一个记录的原来的指针。即可。

还有一种多重链。单向链中每个记录只有一个指针指向下一个记录。多重链中每个记录可以有几个指针。一种常用的多重链是双向链。

双向链是在单向链中增加一个指针,指向前面一个记录。因此,可以向前或向后访问链表。而单向链只允许向后访问链表。双链表比单项链表更方便,但也要用更多的代码。双向链表需要用指针保存表的头和尾。在链表上增加一个新记录时,必须知道第一个记录,最后一个记录,当前记录和当前记录前面一个记录的位置。因此使用双向链表要比单向链表做更多的工作。下面的这段程序说明了如何建立一个双向链表:

```
If FirstCust = Nil Then  
  begin  
    New(CurrentCust);  
    CurrentCust^.Next := Nil;  
    CurrentCust^.Prev := Nil;  
    FirstCust := CurrentCust;  
  end  
Else  
  Begin  
    Prev(CurrentCust);  
    EnterData;  
    PrevCust^.Next := CurrentCust;  
    CurrentCust^.Next := Nil;  
    CurrentCust^.Prev := PrevCust;  
    LastCust := CurrentCust;  
  end;
```

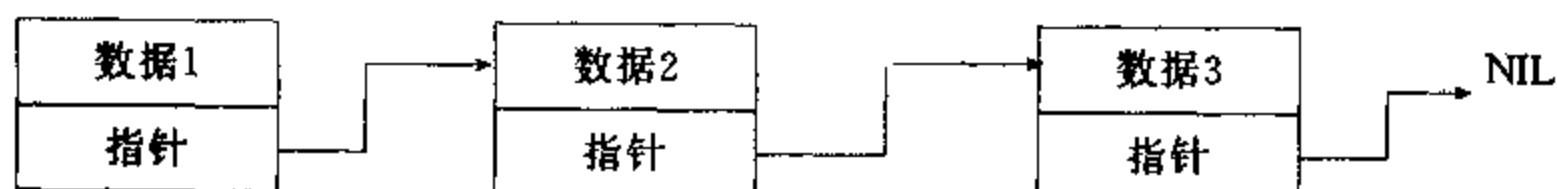
当新的记录是第一个链时,把这个记录的指针 prev 和 Next 都设置为 Nil。另

外几个位置指计 Firstcust, Lastcust 和 prevcust 都设置为指向 Currentcust。当 Firstcust 不是 Ni 时, 程序首先使 prevcust 等于 lastcast, 然后建立一个新的链, 使 prevcust 的 Next 指针指向这个新链, 新链中的 prev 指针指向 prevcust。这样建立起来的双向链可以在两个方向上处理。

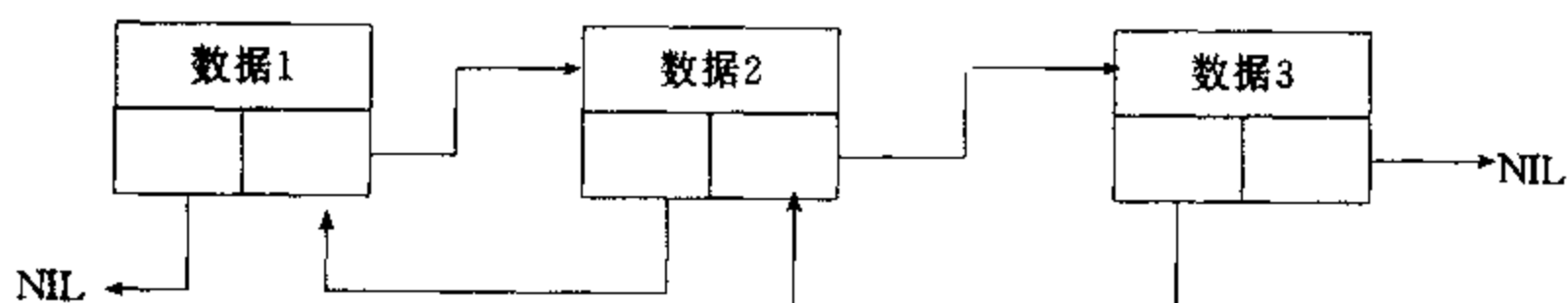
## § 5.4 树

下面介绍一种特殊的多重链表—树, 树有一个头称为根, 但可以有几个尾, 称树顶。树中的每个元素都有两个指针, 分别指向两个不同的元素。

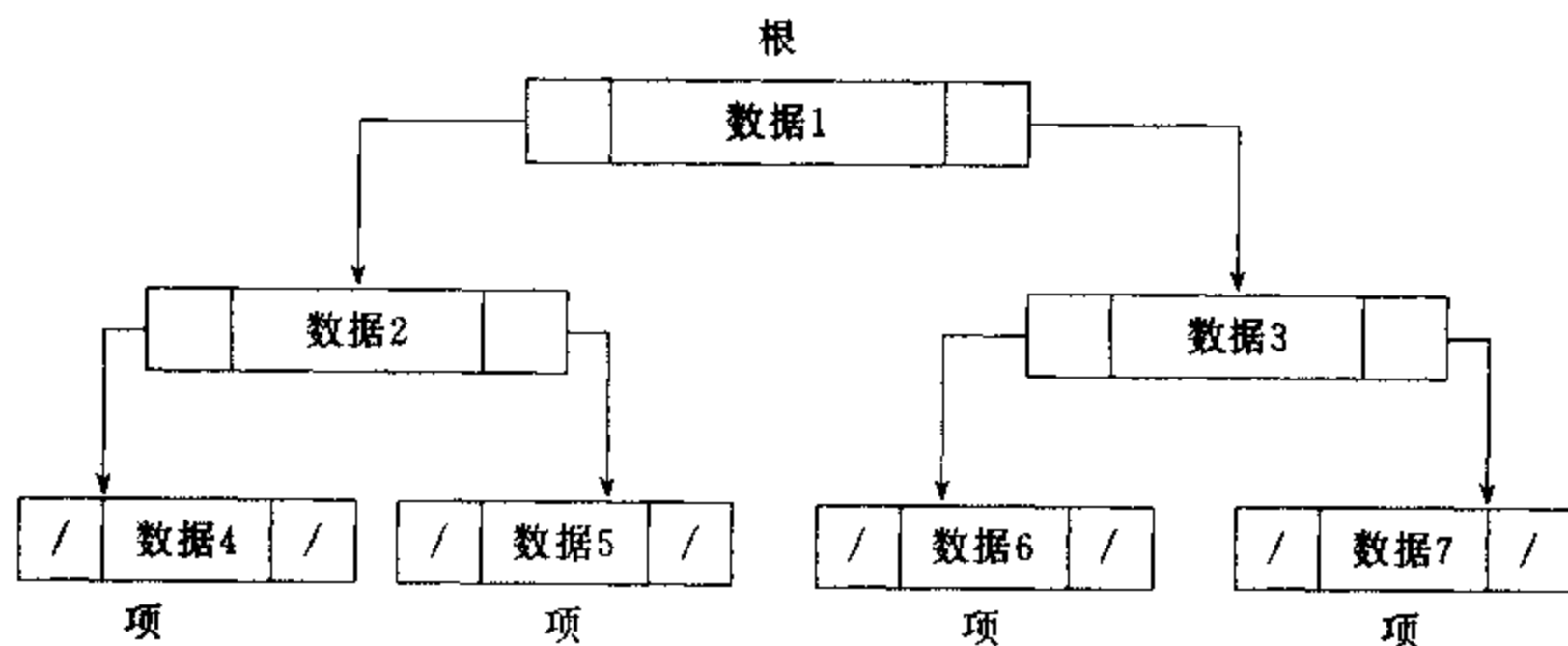
单向链表的结构是每个元素中只有一个指针指向下一个元素。其结构如下图:



双向链表的结构是: 每个元素有两个指针, 一个指向前一个元素, 另一个指向后一个元素。其结构如下图:



树的结构是, 每个元素也有两个指针, 但是这两个指针都是指向其后面的元素, 这两个指针分别称为左指针和右指针。其后面的元素分别称为左后继与右后继元素。其结构如下图:



在树中没有前继元素的只有一个, 也就是树根。没有后继元素的是树顶, 当元素的两个指针都为 Nil 时, 这个元素就是树顶。树中的每个元素都可以看作其子树的根。因为每个元素有两个分枝, 因此可以产生两个子树, 一个左子树, 一个右子树。

用树组织有关的数据，使其成为层次数据结构。可以很方便地进行查找。将树的元素按某种顺序排列，使其左子树的元素排在这个元素之前，右子树的元素排在这个元素之后。在进行查找时，如果关键字的值排在当前结点之前时，只需查找左分枝；否则只需查找右分枝。

下面的程序表明如何建立树：

```

Program createTree;
Type
  Branch = ^ Tree
  Tree = Record
    Word: string[20];    Left, Right: Branch;
  End;
Var
  Nextword: String[20];
  Root: Branch;
Procedure Addtree(Nextword: String[20]; VAR P: Branch);
Begin
  If P = Nil then
    Begin
      New(P);
      With P^ Do
        Begin
          Word := Nextword;
          Left := Nil;
          Right := Nil;
        End
      End
    End
  Else With P^ Do
    IF Nextword < word Then Add Tree (Nextword, Left)
    Else Addtree(Nextword, Right)
  End;
Begin
  Root := Nil;
  ReadLn(Nextword);
  While Nextword <> 'zzzzz' Do
    Begin
      AddTree(Nextword, Root);
      readln(NextWord)
    End
  End
End

```

这个程序首先使 root := Nil。然后接受输入的字符串到 Nextword，最后使用过程 Addtree 将 Nextword 放到适当的位置。在建立树的过程中，如果 Nextword 大于结点的字符串，则向



右子树继续寻找;如果小于,则向左子树寻找,直到一树顶,把它放到新建立的元素中。然后接受下一个字符串输入,直到输入一个'ZZZZZ'的字符串为止。

在处理树中的数据时,需要访问树中的每个元素,即遍历树中的所有结点。下面介绍一种称为中序遍历的遍历方法,中序遍历可以用递归方式描述为:

1. 遍历左子树。
2. 访问当前结点。
3. 遍历右子树。

下面的过程就是中序遍历的递归过程。

```
Procedure Inorder(P;branch);  
  Begin  
    If P <> Nil Then;  
      Begin  
        Inorder(P^.Left);  
        (处理当前结点)  
        Inorder(p^.Right);  
      End;  
    End;
```

在调用 Inorder 过程时,首先使 P 指向树的根 Root。除了中序遍历外,还可以用前序历,即首先访问当前结点,然后遍历左子树,遍历右子树。或用后序遍历,即首先遍历左子树,遍历右子树,然后访问当前结点。

## § 5.5 @ 操作符

在进行地址操作时,经常要把一个变量或一个过程的地址赋给指针。这可以用@操作符完成。@操作符返回其后面标识符的地址,。

例如,若 A 是个整型变量,@A 是这个整型变量的内存地址。下面的程序表明如何用@操作符和指针:

```
Program AddressTest;  
Type B-Type = Array[1..2] of Byte;  
Var  
  i: word;  
  b: ^ B-Type;  
Begin  
  i := $FFFF;  
  b := @i;  
  Writeln(b^[1], ', ', b^[2]);  
End;
```

在这个例子中, b 是指向两个字节的数组, i 是一个字型变量。首先初始化 i = FFFFh, 然后把 i 的地址赋给指针 b。使用指针变量, 就可以分别处理字型变量的两个字节。

## 第六章 文件

计算机的程序和数据通常以文件的形式存贮在磁盘上。现在的计算机一般都至少有一个软盘驱动器，有的还带有硬盘。磁盘可以长期保存大量数据。Turbo Pascal 有强大的磁盘文件操作，帮助程序处理数据。Turbo Pascal 支持三种基本文件类型：文件文件，类型文件和无类型文件。

所有的 Turbo Pascal 文件，无论是哪一种类型的，都可以用作输入和输出。所谓输入，就是说程序可以从文件中取得数据并在程序中使用这些数据。输出就是把程序的结果存到文件中。一个文件即可以用作输入，同时作为输出。Dos 要求每个文件名为一到八个字符，并且可带三个字符的扩展名，用以说明文件的内容。例如，Turbo.exe 是 Turbo Pascal 的程序文件，exe 的扩展名表明这是一个可执行文件。在 DOS 提示符下，键入文件名即可以执行该文件。Turbo Pascal 的源程序文件一般带 .PAS 的扩展名。如果文件存贮在 DOS 目录下，则目录的路径也是文件名的一部分。DOS 文件名的有关规则可查阅 DOS 用户手册。在使用 Turbo Pascal 之前，应首先熟悉这些规则。

### § 6.1 文本文件

文本文件由一些行组成，行中包含字符，词和句子，每行由回车和换行符结束。文本文件中除回车和换行符外，都是由可显示字符组成。而非文本文件可包含任何字符，在屏幕上显示文件文件时，是以可读形式显示的。但非文本文件不能在屏幕上正常显示，会出现一些乱七八糟的东西。

在使用文本文件之前，应先在程序中说明一个文本文件标识符。文本文件标识符的说明与变量一样，但应使用 Turbo Pascal 的保留字 text。在说明了文本文件标识符后，应把这个标识符赋与一个磁盘文件。例如：

```
Var txtFile :TEXT;  
Assign(TxtFile, 'TEXT.DAT');
```

上面的第一行说明 txtFile 为 TEXT 型变量。第二行把 TxtFile 赋与磁盘文件 TEXT.DAT。此后，程序不再使用文件名 TEXT.DAT，程序对 TxtFile 的所有操作都是对文件 Text.DAT 做的。

在把一个文件标识符赋予一个文件后，还应对文件作一些准备工作。这可使用下列三个 Turbo Pascal 命令之一，这三个命令是 Reset, REwrite 或 Append。

当准备用文件作为输入时，用 reset 打开文件。这时只能使用输入命令，否则将产生 I/O (输入/输出) 错误。Reset 命令将文件指针置于文件的开头，所有的输入都是从文件头开始向后进行。如果试图用 Reset 命令打开一个不存在的文件时，也将产生 I/O 错误。使用编译指令 {\$I-}，可以消除这个错误。

当准备用文件作为输出时，应使用 Rewrite 和 Append 命令打开文件。这两个命令都是为输出而打开，但功能却不同。

当用 Rewrite 打开文件时, 文件指针置于文件的开头。如果文件已存在, 则文件的内容将被删除。如果文件不存在, 则建立 Assign 语句中指定的文件。

当文件用 append 命令打开时, 文件的指针将置于文件尾, 所有输出的类据都将附加在已有数据的后面。若试图用 append 命令打开一个不存在的文件时, 将产生一个 I/O 错误。

在文件操作完之后, 必须关闭文件。Close 命令用于关闭文件。这条命令首先把临时缓冲区中的数据存入磁盘, 然后释放一个 DOS 文件柄。文件柄是 DOS 提供给程序用于管理文件操作的。当用 Reset 等命令打开文件时, DOS 把一个文件柄分配给 Turbo Pascal。由于 DOS 限制文件柄的数量, 只能同时打开十五个 Turbo Pascal 文件。最后, Close 命令修改 DOS 的文件目录。

文件一旦关闭, 不能再对文件作输入或输出操作, 除非再次用 reset, RewriteAppend 命令打开。文件关闭后, 文件标识符和磁盘文件之间的联系依然存在。因此, 再次打开文件时, 不需要重复 Assign 命令, 下面的例子说明了这一过程。

```
Program FileTime;  
Var TxtFile:Text;  
Begin  
Assign(TxtFile, 'TEXT.DAT');<---建立 TxtFile 与 TEXT.DAT 之间的联系  
Reset(TxtFile);<-----为读打开文件;  
rewrite(TxtFile);<-----为写打开文件;  
Close(TxtFile);<-----关闭文件  
Append(TxtFile);<-----为追加再打开文件  
Close(TxtFile);<-----最后关闭文件  
End;
```

文本文件用 Reset 命令打开后, 就可以用 read 或 Readln 语句从文件中取得数据。例如  
ReadLn(txtFile,s);

当 txtFile 已经赋予 'TEXT.DAT', 并用 Reset 打开之后, 这条语句从文件的当前行中读出字符, 放到字符串变量 S 中。读完字符后, 文件指针跳过本行中剩下的字符, 指向文件中的下一行。当用 ReadLn 从文件中读一个字符串时有三种可能; 行中字符的个数恰好等于字符串的最大长度; 行中字符的个数少于字符串的最大长度; 行中字符的个数大于字符串的最大长度。在前两种情况下, Turbo Pascal 读出行中的所有字符, 放入字符串中, 并把文件指针移到下一行的开头处。字符串长度等于读入的字符个数。在第三种情况下, Turbo Pascal 读出字符串所允许的最大字符个数, 并把文件指针移到下一行。行中未读的字符将丢失。

Read 语句的操作这程与 Readln 相同, 只是在读完一个字符串之后, 文件在所读的最后一个字符的后面, 不移到下一行的开始处。当 Read 遇到回车/换行时, 停止读字符, 但是文件指针并不向前移动, 直到使用 readln 为上。

使用一个 Readln 语句可以用时向几个字符串输入字符。例如

```
Readln(TxFile, s1, s2, s3);
```

从当前行中读出字符。顺序放入字符串 s1,s2,s3中。如果当前行如下:

```
This is a line of characters.
```

这三个字符串都为 String[5]型, 则字符串的赋值方式如下:

s1 s2 s3 未用到

┌──────────┐  
This is a line of characters

文本文件不仅可以存贮字符，而且也可以用于存贮数值数据。数值文本文件中是按字符形式存贮的，不是按其二进制方式存贮。当从文本文件中读数据时，Turbo Pascal 将跳过空格字符，然后读入字符，直到遇到另一空格字符，或者 CR/LF 为止。这些字符先组成字符串，然后转换成整型数或实型数，这要根据接受数值的变量类型而定。如果转换成功，则把这个值赋给变量；如果不成功，则产生一个 I/O 错误。

如果从文本文件中读出的数字格式不正确时，也将产生一个 I/O 错误。如对整型数输入实型。当整型变量输入大于 32767 的数字时，也将产生 I/O 错误。

由于文本文件具有行结构，可以使用标准 Turbo Pascal 函数 Eoln 测试是否到了行结束的位置。当文件指针遇到回车时，Eoln 返回布尔值 True，否则返回布尔值 false。当文件指针遇到文件尾时，Eoln 也返回 True。另一个标准 Turbo Pascal 函数 eof 是专门用于测试文件指针是否指向文件尾。如果是，则 Eof 返回布尔值 True，否则返回布尔值 False。

使用这两个标准函数可以控制什么时候停止读文件。除了这两个标准函数外，Turbo Pascal 还提供了另外两个标准函数，以提供更完善的文本文件控制。这两个标准函数是 SeekEof 和 seekEoln，与 Eof 和 Eoln 相同。在到文件尾时，Seekeof 返回 True。同样在行结束时，seekEoln 返回 true。但这两个函数有其独特的功能，在测试文件尾或行尾时，可以跳过 0 到 32 个 ASCII 字符，这个范围内的字符可以是空格字符或者标准 ASCII 控制字符。因此，当文件中还有空格字符和其它控制码时，seekeof 也将返回 true。

如前所述，当文本文件用 Rewrite 或 Append 命令打开时，可以用作输出。这可以用 Write 或 Writeln 向文本文件写数据。其格式为

writeln(<文件标识符>,<表达式>);

文件标识符指明后面的数据应送到什么地方。

例如：

WriteLn(TxtFile, name);

这个语句把 name 的内容输送到 TxtFile 指明的文件中，当用 Write 或 Writeln 进行输出时可以使用特殊格式。例如，在参数后加冒号，表明这个值应右对齐。冒号后的数字表明这个数据所占的域宽。例如：

Name := 'Johnson';

WriteLn(Name);

WriteLn(Name:20);

WriteLn(Name:4);

其结果如下：

Johnson

Johnson

Johnson

第一个 writeLn 表达式未使用格式，因此结果是左对齐。第二个语句指明产生一个 20 个字符宽的域。在这个域中，数据右对齐。因此在左也产生了 13 个空格字符。第三个语句虽然也只指明了格式，但是数据的长度大于给定的域值 4，指以这条语合的格式不起作用。

整型的输出格式与字符串的格式相同。由冒号和域宽数字组成，在这个域中数字将右对齐。实型和格式中可带一参数，用于指定域宽。也可以指定两个参数，第一个参数用于指定域宽，第二个用于确定小数的位数。例如：

```
r:=123.23;
writeln(r);           (* 结果: '1.2323000000E+02' *)
WriteLn(r:0);         (* 结果: '1.2E+02'      *)
Writeln(r:10);        (* 结果: '1.2323E+02'    *)
WriteLn(r:10:2);      (* 结果: '123'          *)
WriteLn(r:0:0);       (* 结果: '123'          *)
```

Program Count;

Var

F: Text;

ch: char;

Line, Chnum: Integer;

Begin

Assign(F, 'TEXT.DAT');

Reset(F);

Line:=0;

Cjmin:=0; While Not Eof(F) Do

Begin

While Not Eoln(F) Do

begin

Read (F, ch);

Chnum:=Chun+1;

End;

redLn(F);

Line:=Line+1;

End;

WriteLn('LINE=', Line, 'CHNVM, Chuwum');

End'

下面的一段程序统计文件中的字符个数，以及有多少行。

这段程序用 Eoln 测试一行是否结果。

## § 6.2 类型文件

类型文件是由特定类型数据的文件，如整型，实型，记录等等。这种文件可以使编程更加简洁有效。而且类型文件的输入输出要比文本文件快得多。类型文件有严格的结构。类型文件一般按下列方式说明：

type

<文件类型标识符>=file of <类型>;

var

<文件标识符>: <文件类型标识符>;

也可以直接说明为:

var

<文件标识符>: File of <类型>;

其中 File 是 Turbo Pascal 的保留字, 它说明变量是文件类型。类型是文件中各分量的类型, 可以是除文件之外的任何数据类型。文件是由相同类型的分量构成, 各分量按其建立的先后顺序存放在文件中。例如下语句说明标识符 f 为一实型文件:

Var

f: File of Real;

这条语句说明这个文件只用于存放实型数。在这个文件中存放的实数格式与计算机内存中存储形式是一样的。在读文件和写文件时, 不需要转换, 可以直接传送到内存或文件中。因此, 处理速度比文本文件快得多。

与文本文件不同, 类型文件不是由行组成的。因此不能使用 Readln 和 Writeln 语句。类型文件是由记录组成的, 每条数据项对应一条记录。由于 Turbo Pascal 中实型数需要6个字节, 所以实型文件中每个记录的长度也是6个字节。文件中的前6个字节组成第一个记录, 接下来的6个字节组成第二个记录, 依次类推。再如, 整型数只需两个字节, 在类型文件中, 每个记录也只有两个字节。例如下面这段程序向 Real.DAT 文件写入三个实型数, 由于每个实型数需要六个字节, 所以这个文件占十八个字节。这可以在 DOS 提示符下用 Dir 命令证实。

Var

r: Real;

f: file of Real;

Begin;

Assign(f, 'REAL.DAT');

Rewrite(f);

r := 100.234;

Write(f, r);

Write(f, r);

r := 32.23;

Write(f, r);

r := 9849.40;

write(f, r);

Close(f);

end;

类型文件可以是任何类型的类据。因此如可以是字符串型的文件。字符串型文件和文本文件虽然都是由字符组成, 但是它们的存储方式是不同的。文本文件是由行组成, 而字符串型文件是由记录组成。例如:

type

s10 = string[10];

Var

```

    txtFile:Text;
    stringFile:File of s10;
    S:s10;
Begin
    Assign(txtFile, 'OUT.TXT');
    Rewrite(tXTFile);
    Assign(StringFile, 'OUT.STR');
    Rewrite(sringFile);
    s:='ABCD';
    WriteLn(TxtFile,s);
    Write(StringFile,s);
    Close(TXtFile);
    Close(StringFile);
End

```

这段程序对文本文件 OUT.TXT 和字符串型文件 OUT.STR 中写入字符串 'ABCD'，在文本文件存贮文方式如下：

A B C D

而在字符串型文件中存贮方式如下：

4 A B C D X X X X X X

在文本文件中只用四个字节。而在字符串型文件中用了 11 个字节。第一个字节是实字符串的长度，接下来的 4 个字节是字符串，后面六个字节未用到，但其它文件也不能占用。字符串型文件要占更多的空间，它要有长度字节和一些无效字节。

对于用户自己定义的数据类型，也可以定义存贮这类数据的文件。例如：

```

Type
    CustomerRec=Record
        Name:string[30]; Address:String[40];
        Telephone:String[15];
    End
Var
    Cust: CustomerRec;
    CustFile: File of custermerRec;
Begin
    Assign(CustFile, 'Cust. Dat');
    Rewrite(CustFile);
    With cust Do
        Begin
            Name:='L. M. Quibble';
            Address:='New york City'; Telephone:='(123) 047-7580'; End
        Close(custFile);
    End.

```

在这段程序中定义了一个记录类型 Cust, 包含有 Name, Address, Telephone 等域。同时定义了存贮这类记录的文件 Cust. dat。由于这个文件为相应的记录类型的文件, 所以使用一条语句就可以把一条完整的记录读写完成, 不必分别读写每条记录的各个分量。因此处理速度也就加快了。

### § 6.3 无类型文件

Turbo Pascal 提供的另一种文件是无类型文件。文本文件由行组或; 类型文件由特定的数据结构类型组成。而无类型文件不限制数据结构的类型。可以从无类型文件向任何数据类型读数据。由于可以直接从磁盘向数据结构传送数据, 无类型文件可以用于高速输入输出。

在用 reset 或 Rewrite 语句打开无类型文件时可以指定第二个参数, 用于指定记录的大小。例如:

```
Reset(SFile, 1);
```

这条指令为读而打开文件, 定义记录的长度为1个字节。如果对于整型数, 应定义记录的长度为2。Turbo Pascal 不要求使用的数据类型的大小与记录的长度匹配, 指定记录长度只是为方便编程。如果不指定记录长度, 其缺省值为128个字节。无类型文件的说明方式如下:

<文件变量> :File;

无类型文件的读写操作应使用两个标准 Turbo Pascal 过程:BlockRead 和 BlockWrite。其格式如下:

BLOCKREAD(<文件变量>, <变量1>, <字节数>, <变量2>);

第一个参数是文件标识符, 用于指明应从哪个文件中读数据。第二个参数是存放数据的数据结构, 例如字节数组等。第三个参数用于指明一次读操作所能读入的最大字节数。可以直接填写所需字节数, 也可以使用 Turbo Pascal 提供的标准函数 Sizeof, 这个函数返回指定的数据结构所占用的字节数。第四个参数为整型变量, 用于记录实际读入的记录个数。当文件中剩下的数据不是最大读入记录数时, 实际读入的记录个数会少于第二个参数指定的值。

Blockwrite 的格式与 Blockread 相似, 但只有三个参数: 第一个是文件标识符, 第二个是用于输出的数据结构, 第三个是输出到文件中的记录个数。下面的程序说明如何使用无类型文件, 它把一个文件拷贝到另一个文件中。

```
Program CopyFile;
Uses CRT;
Var;
    SourceFile
    DestFile : File;
    RecrdRead : Integer;
    Buffer : Array [1..1000] of Byte;

Begin
    CluScr;
    If ParamCount <> 2 Then
        Begin
```



```

    WriteLn('CopyFile [FromFile] [ToFile]');
    Halt;
    End;

Assign(SourceFile, ParamStr(1));
{SI-}
Reset(SourceFile, 1);
If IOresult <> 0 Then
    Begin
        WriteLn(ParamStr(1), ' Not found. ');
        Halt;
        End;

Assign(DestFile, ParamStr(2));
Rewrite(DestFile, 1);

WriteLn('. = 1,000 bytes copied. ');
BlockRead(SourceFile, Buffer, SizeOf(Buffer), RecordRead);
While RecordsRead > 0 Do
    Begin
        Write('. ');
        BlockWrite(DestFile, Buffer, RecordsRead);
        BlockRead(SourceFile, Buffer, SizeOf(Buffer), RecordsRead);
        End;

Close(SourceFile);
Close(DestFile);
WriteLn;
Write('Press ENTER... ');
ReadLn;
End.

```

类型文件和无类型文件都可以称为随机存取文件，即文件中的记录可以不按顺序存取，例如，可以先读第3个记录，然后读第12个记录，再读第2个记录，等等。这时，首先使用 seek 语句使文件指针指向指定记录，然后用 Read 语句将相应的记录读出。例如下面语句把文件中的第3个记录读出。

```

Seek(CustFile, 2);
read(CustFile, Cast);

```

对第三个记录，Seek 语句中用的是2，因为在 Turbo Pascal 中类型文件从0记录开始。在随机读记录时，还可以使用另外两个标准 Turbo Pascal 函数 Filesize 和 FilePos。Filesize 返回文件中记录的个数，FilePos 返回当前文件的指针。同样，也可以按任何顺序改写文件中任一个记

录。首先用 Seek 语句找到相应的记录,再用 Write 语句把相应的数据写入文件。

应该注意的是,对于非文本文件,即类型文件和无类型文件,只要用 reset 命令打开一个已存在的文件,就可以随时读或写记录,不必再用 Rewrite 命令打开。

## § 6.4 缓冲区

文件一般都存放在磁盘上,每次读或写文件都要使用磁盘,这会大大降低程序运行的速度。因为磁盘操作是非常慢的,在读或写磁盘时,首先要确定数据的位置,然后把磁盘的内容读入计算机内存,或把内存中的数据写到磁盘上。这要用到很多机械操作,对于计算机来说,这种操作是非常慢的。为了提高程序的效率,减少磁盘操作,可以在内存中保留一小块内存区域用作磁盘操作。这一块内存区域称作缓冲区。使用缓冲区可以不必每次读或写磁盘都实际进行磁盘的读写。例如下面这段程序。

```
read(txtFile,ch1);  
Read(txtFile,ch2);  
Read(txtFile,ch3);  
Read(TxtFile,ch4);  
Read(TxtFile,ch5);
```

从一个文本文件中读出五个字符。如果不使用缓冲区,每个字符都要实际从磁盘中读一次,若使用缓冲区,在读第一个字符时,Turbo Pascal 一次从磁盘中把所有五个字符都读到缓冲区中,后面的四条语句的读可以从缓冲区中取出,不必再实际读磁盘。因而减少了磁盘操作。

Turbo Pascal 为文本文件提供了一个128字节的缓冲区。这样,程序在读磁盘上的文本文件时,每次都都要读128个字节的数据。不论要读多少个字符,即使只读一个字符,在第一次读磁盘时也要读入128个。但是其余的字符由 Turbo Pascal 管理,不需要程序干预。

在处理一个很大的文本文件时,只使用标准的128字节缓冲区就显很不够,便用 Turbo Pascal 提供的过程 SetTextBuf 可以定义一个更大的缓冲区。例如:

```
Var  
f:text;  
Buffer:array[1..512] of Byte;  
Begin  
Assign(f,'TEST.DAT');  
SetTextBuf(f,Buffer);  
Reset(f);
```

这段程序为文本文件 'TEST.DAT' 设置了一个512个字节的缓冲区。根据需要,缓冲区还可以扩得更大、要注意的是,调用过程 SetTextBuf 必须在打开文件之前,否则有可能丢失数据。缓冲区应该是全局性的,如果把缓冲区说明为局部的,当释放局部变量时有可能丢失一部分数据。

使用缓冲区可以减少实际磁盘操作的次数。向带缓冲区的文件写数据时,Turbo Pascal 实际上是把数据送到缓冲区。当缓冲区满了之后,再把缓冲区的所有内容一次写到磁盘上,同时清空缓冲区,以备接着输出。在缓冲区填满之前,要想把缓冲区的内容记入磁盘,可以使用 Turbo Pascal 提供的过程 Flash。Flash(f)语句把 f 的缓冲区内容立即记入磁盘,这样就不会

丢失数据。关闭文件会自动把缓冲区的内容记入磁盘。

## § 6.5 文件的删除与改名

Turbo Pascal 提供的文件服务还有修改文件名和删除文件。只要使用 Turbo Pascal 提供的两个过程 `rename` 和 `Erase`，可以在程序中修改文件名和删除文件，而不必退出到 DOS 提示符下。

要修改文件名，首先要将文件指定一个文件变量，然后调用 `rename` 修改文件名。例如下面两条语句把 'File. OLD' 文件改名为 'FILE. NEW'。

```
Assign(f, 'FILE. OLD');
```

```
Rename(f, 'FILE, NEW');
```

删除文件的方式与修改文件名的过程一样。首先将文件指定一个文件标识符，然后调用 `Erase`。如：

```
Assign(f, 'FILE. TXT');
```

```
erase(f);
```

下面这段程序演示了如何使用 `Rename` 和 `Erase` 过程对磁盘文件改名或删除。当程序启动后，提供了三个选择：按 R 键对文件改名。这时要求输入原文件名和新文件名。程序将把原文件的名字改为新文件名。按 E 键删除文件。程序将要求输入要删除的文件名。若按 Q 键，则停止程序的执行。

```
Program FileOperate;  
Use CRT;  
Var  
  F1: File;  
  N1, N2: String[255];  
  Ch: Char;  
Begin  
  Clrscr;  
  Repeat;  
    Write('R)rename, E)rase, Q)uit:');  
    ReadLn(ch);  
    Case Upcase(ch) of  
      'R':  
        Begin;  
          Write('Name of file to rename:');  
          ReadLn(N1);  
          Write('New name for the file:');  
          ReadLn(N2);  
          Assign(F1, N1);  
          Rename(F1, N2);  
        End;  
    end;
```

```
'E'  
  Begin  
    Write('Name of file to erase;');  
    ReadLn(N1);  
    Erase(F1);  
    End;  
  End;  
  Until Upcase(ch) = 'Q';  
End
```

## 第七章 外部过程，过程与函数库

作为一种高级语言，Turbo Pascal 有强大的功能和很高的速度。但是有一些计算机的特殊功能，高级语言不能充分利用。而汇编语言可以充分利用计算机的各项功能，并且汇编语言执行起来效率更高。

因此，尽管 Turbo Pascal 语言有很多优点。用 Turbo Pascal 语言编写程序更方便，速度更快，移植也比较方便。但是，为些一专门目的，使用汇编语言。将汇编语言编写的程序加到 Turbo Pascal 程序中，这样得到的程序即具有 Turbo Pascal 的逻辑结构，又有汇编语言的高速度。

在 Turbo Pascal 程序中插入汇编语言的方法有两种：嵌入代码方式和外部过程方式。外部过程方式是把汇编语言程序编译成 OBJ 文件，在链接时，与 Turbo Pascal 程序链接一起。代码嵌入方法是在 Turbo Pascal 程序中直接插入机器语言指令。Turbo Pascal 不检查外部过程或嵌入代码的错误，因此在使用时要非常小心，对这些代码要进行充分的调试。

### § 7.1 嵌入代码

嵌入代码是在 Turbo Pascal 程序中插入机器指令。这要求对汇编语言和 Turbo Pascal 语言都非常熟悉。在计算机刚出现时，没有任何程序设计语言。所有的程序都是按“机器语言”的形式直接输入到计算机中的。机器语言用代表指令的数码组成。用这种方式编写和维护程序是非常困难的。

后来产生了汇编语言。汇编语言使用的是助记符而不是数码，因而比机器语言方便。但是用汇编语言编写程序很复杂，而且容易出错。

因而出现了各种高级语言。一条高级语言的语句可以产生许多条汇编语言的指令，所以编写程序的速度大大提高了。而且高级语言的移植性好，可以很容易从一种机器搬到另一种机器上。这样，高级语言得到了广泛的应用，汇编语言只用于很少的一些专用程序。

嵌入代码就是在 Turbo Pascal 程序中插入代表机器指令的数码，即最原始的编程方式。Turbo Pascal 的编译指令 Inline 指明后面的是机器指令。机器指令为由“\$”符号开始的十六进制

FE); (\* MOV [BP-2], AX \*)

这段代码是将变量 i 放到寄存器 AX 中，再把 AX 中的值加上变量 j，然后将结果放到 [BP-2] 指定的单元，即堆栈中的某个位置中，其中每条机器语言的意义在每行后面的注释中说明。在这个例子中虽未明确说明，i 和 j 都应看作是堆栈中的值参数。因为若使用全局变量，必须使用其它的嵌入代码。在这个例子中使用了一个专用的操作符 <，它表示后面的变量仅取一个字节。另外一个操作符 > 表示后面的变量为一个字。对于不同的变量，正确使用 <

和>这两个操作符是非常重要的。如果变量是字节型的，或者需要一个字节一个字节地处理数据，应使用<操作符。如果变量是字型的，或需要一个字一个字地处理数据，应使用>操作符，否则可能产生错误的结果。要确保正确使用宽度的唯一办法是用一个调试器，例如 Turbo Debugger，将嵌入代码反汇编，查看其形式是否正确。

为了调试方便，嵌入代码最好使用下面的形式：

```
Inline ($ 8B/$ 46/<i/);      (* MOV AX, I *)
```

```
Inline($ 03/$ 46/<j/);      (* ADD AX, J *)
```

```
Inline($ 89/$ 46/$ Fe/);    (* MOV [BP-2], AX *)
```

这种方法虽然比较烦，但调试起来比较容易。因为在第一种方法中，所有的嵌入代码都在一个分为三行的语句中。当用 Turbo debugger 反汇编列表时，只有第一行才出现在反汇编列表中，另外二行不会列出来。但在第二种方法中每行作为一条语句，在用 Turbo Debugger 反汇编列表中，每行都会列出来。因而更容易调试。

嵌入代码也可以为函数，即嵌入函数，嵌入函数与一般的 Turbo Pascal 函数一样。有一个函数头，用 Function 开始，后面是函数名和参数表。还有一个函数体，用 Begin 开始，由 End 结束。

函数体中的语句都是用 Inline 开始的机器语言指令，下面是使用两个数相加的嵌入函数的例子：Program TestIntine;

```
Uses CRT;
```

```
Function Sum(i,j:Integer): Integer;
```

```
Begin
```

```
Inline($ 8B/$ 6/<i/);      (* MOVE AX, I *)
```

```
Inline($ 03/$ 46/<j/);      (* ADD AX, J *)
```

```
Inline ($ 89/$ 46/$ FE);    (* MOV [bP-2], AX *)
```

```
End;
```

```
Begin
```

```
Clrscr;
```

```
Writ('1+2=');
```

```
writeln(Sum(1,2));
```

```
End;
```

在这个例子中，每个机器指令用十六进制形式输入，之间用分开。每行为一条语句。上面这个函数反汇编的结果如下：

PUSH	BP	
MOV	BP,SP	
SUB	SP,+02	
MOV	AX,[BP+08]	嵌入代码
ADD	AX,[BP+06]	
MOV	[BP-02],AX	
MOV	AX,[BP-02]	
MOV	SP,BP	
POP	BP	

在反汇编列表中,前后共增加了七条指令。前三行建立过程入口堆栈,其中的前二行对 Turbo Pascal 过程和函数都是相同的。第三行 `Sub sp, +02` 是为返回结果保留位置,即 Turbo Pascal 在堆栈中保留了二个字节,用于临时存放结果。最后四行是从堆栈中把结果放入 AX 中,恢复 SP 和 BP,然后返回调用处

只有中间的三行是原来的嵌入代码。

Turbo Pascal 要求返回的结果放在堆栈中的固定地方,因此在嵌入过程中返回函数值是需要一些技巧的。在过程结束之前,必须保证返回的函数结果放在堆栈中的正确位置上。

在这段程序的结束处,返回是个 RETF。这是因为在程序编译时使用了 `{F+}` 编译指令。因此,在编写一个嵌入过程和函数时,不仅要知道机器语言是如何工作的,而且还要了解 Turbo Pascal 在汇编方面是如何工作的。

使用嵌入函数和过程不如使用外部汇编过程容易。外部汇编过程可以先汇编成 OBJ 文件,再用 External 指令与 Turbo Pascal 程序链接到一起。但有时还是要使用嵌入代码。这就是嵌入指令。

嵌入指令与嵌入函数基本相同,但是省掉了 Begin 和 End 关键字。因此 Turbo Pascal 不为嵌入指令建立或清除指令。下面是一段。嵌入指令。可以说明嵌入指令与嵌入函数有什么不同:

```
Function sumD(i,j:Integer):Integer;
Inline ($ 581          (* POP AX *)
      $ 5B/           (* POP BX *)
      $ 03/$ C3);      (* ADD AX,BX *)
```

当它编译出来后的结果为:

```
POP AX
POP BX
ADD AX,BX
```

可以看出,一条指令也没有增加。但是,对于嵌入指令不能用名字访问变量。因为此时 Turbo Pascal 没有建立堆栈,不能用相对 BP 的偏移量来取参数。这时只能通过从堆栈顶直接弹入寄存器来访问参数。当然,也可以自己建立堆栈,这时可以使用相对于 BP 的偏移量。

用嵌入代码编程序是很不方便的。嵌入代码最好只用于短代码,以完成特定的功能。例如可以用嵌入代码取得 SP 的值,放入一个字型变量中。下面就是这样的一段化码,它把 SP 的值放入变量 WR 中:

```
Inline($ 89/$ 26/WR); (* MOV WR,SP *)
```

当需要时,这是最有效的。在编辑高级程序时,使用这种特殊的编程方法的有很多好处。

## § 7.2 外部过程

外部过程是用汇编语言编写的例程,编译成 OBJ 文件。在链接时,把它与 Turbo Pascal 程序链接到一起。与嵌入代码比较,汇编程序有许多优点。与嵌入代码中使用的机器语言相比,汇编程序更容易编写和维护。更重要的是,外部过程可以直接访问 Turbo Pascal 的全局变量,局部变量,参数,过程和函数。在汇编中访问 Turbo Pascal 的数据和函数就象在 Turbo Pascal 中一样方便。编写外部汇编过程对 Turbo Pascal 是一种很有吸引力的方法,特别是在速

度需要考虑的时候,下面是一个用汇编语言编写的外部例程的例子:

```
CODE    SEGMENT BYTE PUBLIC
        ASSUME CS:CODE
PUBLIC  SUM
SUM      PROC  FAR
        PUSH  BP
        MOV   BP,SB
        MOV   AX,[BP+08]
        ADD   AX,[BP+06]
        POP   BP
        RET   4
SUM      ENDP
CODE     ENDS
        END
```

这个例子与前面嵌入代码的例子相同,是求两个数的和。这段程序虽短,但足以说明应如何编写汇编子程序。首先应把代码段的段名说明 CODE,因 Turbo Pascal 只使用两个段名 CODE 和 CSEG。过程名 SUM 产须说明为 PUBLIC,只有这样,这个例程才能被其它程序模块调用。若不把 SUM 说明为 PUBLIC, Turbo Pascal 将不能调用它。

任何外部过程都应首先保存 BP,设置堆栈指针。随后,从堆栈中取得参数相加,结果放在 AX 寄存器中。Turbo Pascal 规定,凡是返回标量的函数,如字节,整型,字等,结果都应放在 AX 寄存器中。过程结束的指令是 RET 4,数字4 表示 Turbo Pascal 在调用这个例程时,把 4 个字节(两个整型数)的参数放入堆栈。应保证在从外部过程返回时,释放掉堆栈中的所有参数。

要在程序中使用这个外部例程,必须选把它编译成 .OBJ 文件。如果这段汇编代码在 ADD.ASM 中,编译后的目标文件是 ADD.OBJ。在 Turbo Pascal 程序中,可以如下说明这个解部例程:

```
{ $F+ }
{ $L ADD }
Function sum(i,j:Integer):Integer; External;
```

首先用编译指令 { \$F+ } 强制远程调用。则于在外部例程中说明为 FAR PROC,其结束语句为 RETF。必须保证 Turbo Pascal 用远程调用。第二行的编译指令 { \$L ADD } 告诉 Turbo Pascal 在 ADD.OBJ 文件中查找外部调用的外部过程。

最后一行是函数说明,包括一个 External 说明符,它告诉 Turbo Pascal 这个过程的代码在目的文件中。这样说明了一个外部过程后,就可以在 Turbo Pascal 程序中使用这个解部过程了。其方法与使用内部过程一样。例如,在程序中可以同下面这条语句调用外部过程 sum。

```
Writeln(Sum(1,2));
```

在汇编子程序中可以使用 Turbo Pascal 的全局过程,变量和类型常量。因此,可以在程序中灵活地使用 Pascal 和汇编代码,代码的维护更加容易。为了说明汇编程序如何使用全局数据和过程,我们看一个例子。在这个例子中,汇编子程序把字符串中的小写字母变成大写字母。这个子程序接受一个字符串参数,但是希望限制字符串的长度,以免字符串太长,造成死



机,汇编子程序测试字符串的长度,当长度超过限制时,将调用一个 Turbo Pascal 过程.程序中设置一个全局变量 Maxstrlen,用于存放字符串的最大长度.下面是汇编语言子程序:

```

DATA          SEGMENT BYTE PUBLIC
               EXTRN MAXSTRLEN,BYTE
DATA          ENDS
CODE          SEGMENT BYTE PUBLIC
               ASSUME CS:CODE, DS:DATA
               EXTRN STRLENERROR,FAR
               PUBLIC UPCASESTR
UPCASESTR     PROC  FAR
               PUSH  BP
               MOV   BP,SP
               LDS   SI,[BP+6]
               MOV   AL,BYTE PTR[S1];    字符串长度放入 AL
               XOR   CX,CX
               MOV   CL,AL                ;字符串长度放入 CL
               CMP   CL,MAXSTRLEN
               JL    LOOP1                ;如果长度小于最大长度,跳转.
               CALL  STRLENERROR          ;否则,调用 StrlenError
LOOP1:        INC   SI
               MOV   AL,BYTE PTR[S1]     ;取字符到 AL
               CMP   AL,97
               JB    NOTLOW               ;小于 'a' 时跳转
               CMP   AL,122
               JA    NOTLOW               ;大于 'Z' 时跳转
               SUB   AL,32                 ;变为大写字母
               MOV   BYTE PTR[S1],AL     ;放回字符串
NOTLOW:       LOOP  LOOP1
               POP   BP
               RET   2
UPCASESTR     ENDP
CODE          ENDS
               END

```

在这段程序中,当字符串长度大于所给的最大长度时,将调用一个叫 StrlenError 的外部过程,这是一个 Turbo Pascal 全局过程.在这个汇编子程序开始时,首先找到字符串参数的长度字节,把它与限制值 Maxstrlen 比较,若大于或等于这个最大值,将把控制传给 Turbo Pascal 的过程 StrlenError.这个过程将显示出错信息并停止程序的执行.在这个程序中作用的测试值 Maxstrlen 是 Turbo Pascal 的一个常量.下面是整个 Turbo Pascal 程序的列表.

{SF+}

Program TestAsm;

```

Uses CRT;
Const
    MaxStrLen : Byte = 100;
Var
    s : string;

{SL UPCASE}
Procedure UpCaseStr(Var s : String); External;
( * * * * * )
Procedure StrLenError;
Begin
    WriteLn('String legnth error encountered. ');
    WriteLn;
    Write('Press ENTER... ');
    ReadLn;
    Halt;
End;
( * * * * * )
Begin
    ClrSce;
    s := 'abcdef';
    WriteLn('Lower case = ',s);
    UpCaseStr(s);
    WriteLn('Upper case = ',s);
    WriteLn(s);
    WriteLn;
    WriteLn('Force a string—length error condition. ');
    WriteLn;
    Write('Press ENTER... ');
    ReadLn;
    s[0] := Chr(101);
    UpCaseStr(s);
    End.

```

由于这个外部函数定义为远程调用，在编译这个程序时必须使用编译指令{\$F+}。否则当外部过程返回时，程序将崩溃。

在前面的例子中，所有的参数调用都是使用基指针 BP 的偏移量。这种方法很费时间，而且容易出错。为此，引进了 Turbo Assembler。这个编译器是专门为编译 Turbo Pascal 程序使用的汇编子程序而设计的。它在一般的汇编语言编译器中增加了一些功能。使汇编子程序与 Turbo Pascal 的链接更容易。下面举个例子说明如何使用 Turbo Assembler，编译器。这个 Switch 过程交换两个变量的值。这个过程要求 2 个参数；两个指针分别指向要交换的两个变

量;第三个是 Word 型参数,它给出要交换参数的长度。例如整型为2个字节,等。对于一般的汇编子程序,首先要建立数据段和代码段。还要建立堆栈的入口和出口,以及要弹出的参数的个数。在使用 Turbo Assembler 后,这些工作不必用户自己再做了。Turbo Assembler 已经替用户做好了,下面是这个程序的列表:

```

·MODE TPASCAL
·DATA
BUFFER DB 256 DUP(?)      ;用作转换的缓冲区
·CODE
PUBLIC SWITCH
Switch PROC FAR A:DWORD, B:DWORD, Dsize:WORD
;将 A 放入缓冲区
LDS  SI,A      ;A 地址装入 DS:SI
LEA  DI,BUFFER  ;缓冲区地址装入 ES:DI
MOV  CX,D      ;数据长度放入 CX
REP MOVSB BYTE PTR ES:[DI],DS:[SI]
;将 B 放入 A 中
LDS  SI,B
LES  DI,A
MOV  CX,Dsize
REP MOVSB BYTE PTR ES:[DI],DS:[SI]
;将缓冲区内容放入 B 中
LEA  SI,BUFFER
LES  DI,B
MOV  CX,Dsize
REP MOVSB BYTE PTR ES:[DI],DS:[SI]
RET
Switch ENDP
END

```

其中的第一行 MODEL TPASCAL 告诉 Turbo Assembler 产生的代码将与 Turbo Pascal 程序相链接。后面的 DATA 指令代替了其它汇编程序要求的预处理代码。下面的一行

```
Switch PROC FAR a:DWORD, b:DWORD, Dsize:WORD
```

使 Turbo Assembler 知道,这个过程的名字是 Switch,它要用远程调用。这个过程带三个参数:两个地址,即 A 和 b,和一个数字值参,即 Dsize。

在前面的汇编子程序中,不带任何入口和出口代码。即使最后的 RET 指令也不必指出需要从堆栈中弹出的字节数。所有这些都由 Turbo Assembler 代做了。而且, Turbo Assembler 允许用 Turbo-Pascal 中的名字引用参数和全局变量。因此,用 Turbo Assembler 编写外部过程要比用标准汇编语言容易得多。

下面的程序表明如何在 Turbo Pascal 程序中使用汇编程序。编译指令 L 指出要链接的目

标文件的名字。{\$F+};

Program Switch Test;

Uses CRT;

Var

a, b: Integer;

C, d: real;

e, f: string;

{ \$L SWITCH }

procedure switch (Var a, b; c: Integer); External;

Begin

Clrscr;

a:=1;

b:=2;

c:=12.34;

d:=45.67;

e:='ABCDEFGH';

f:='HIJKLMN';

Writeln('Using assembler');

WriteLn;

WriteLn(a, '>', b);

switch(a, b, sizeof(a));

WriteLn(a, '<', b);

WriteLn;

WriteLn(c:0:2, '>', d:0:2);

Switch(c, d, sizeof(c));

WriteLn(c:0:2, '<', d:0:2);

writeLn;

writeLn(e, '>', f);

switch(e, f, size of (e));

WriteLn(e, '<', f);

writeLn;

End.

这个例子只是刚刚触及 Turbo Assembler 的强大能力。

### § 7.3 Turbo Debugger

用汇编语言编写程序的最大困难是调试。如果没有调试器，几乎不可能写汇编程序。调试器的一条主要功能是单步追踪能力。Borland 提供的 Turbo debugger 对编写汇编程序是一种非常有用的工具。使用这个工具可以一次执行一条机器指令，这样可以清楚地看到寄存器和内存中的变化。从中可以了解 Turbo Pascal 在内部是如何工作的：参数是如何传送到堆栈中，运

算是如何完成的等等。

为了说明 Turbo debugger 是如何工作的,看一下在 § 7.1 节中的含有嵌入代码的程序。在编译这个程序时,必须使 standalone debugger 选择项打开,并且使用编译指令 { \$D+,L+ }, 这样告诉 Turbo Pascal 信息,使程序的源代码与可执行代码匹配。在程序编译完并记到磁盘上后,键入 TD 后面跟程序名,即可启动 Turbo Debugger。例如前面的那个程序的名字若为 TEST.PAS, 则命令

```
C>TD TEST
```

将启动 Turbo Debugger, 装入 TEST.EXE 程序并读入 TEST.PAS 和 TEST.MAD 文件, Turbo Debugger 利用 .EXE 文件带的信息,可以在显示一行源代码的同时,显示其对应的机器语言指令。这时屏幕的左侧有一个箭头,指向将要执行的下一条试句。Turbo Debugger 启动后,箭头指向程序中的第一个 begin 语句。

使用 F7 和 F8 键可以一次执行一行试句。但是这两个键还是有区别的。遇到函数调用时, F8 键一次执行完函数调用,而 F7 将进入函数。按键盘右侧的数字小键盘上的光标键,可以使程序上滚或下滚,以查看程序的其它部分。如果下滚了若干语句,然后按 F4 键, Turbo Debugger 将执行所有这些语句,直到当前位置。

要充分利用 Turbo Debugger, 必须进入机器指令水平。只要按 F10 激活主菜单, 选择 View 选择项, 然后按 C 选择 CPU。这时将打开 CPU 窗口。它有四个“窗格”。左上角的窗格显示仅汇编代码。右边的是寄存器窗格, 在其中显示 CPU 寄存器的内容。右下角的是堆栈窗格。左下角的窗格显示的是部分 RAM 的内容。最上面的的代码窗格显示的是 Pascal 语句。例如:

```
TESTINLINE. 20; WriteLn(Sum(1,2));
```

这是源程序中第 20 行, 输出函数 Sum 的结果。这下面是运行语句所需的机器指令, 由三部分组成。左测是指令在代码段中的地址, 中间是用十六进制表示的机器语言指令, 最后是汇编语言代码, 例如:

cs:0059 BF5001	Mov	di,0150
cs:055c 1E	push	ds
cs:005D 57	push	di
cs:005E B80100	mov	ax,0001
cs:0061 50	push	ax
cs:0062 B80200	mov	ax,0002
cs:0065 50	push	ax
cs:0066 E897FF		

这就是前面那条 Turbo Pascal 语句的执行代码。可以看到, 一条语句产生了 14 行机器语言代码, 其中有两处调用了其它过程。程序中的第一个函数调用是调用含嵌入代码的过程 Sum。按 F7 直到 Turbo Debugger 进入 Sum, 这时屏幕将显示与这个过程对应的机器指令代码, 过程开始时的代码为:

```
DX:0000 55      PUSH  BP
```

```
CS:0001 89E5      MOV   BP SP,
```

```
CS:0003 83EC02    SUB   SP,0002
```

这段代码是 Turbo Pascal 加到嵌入函数前面,在过程执行前设置堆栈的。接下来是嵌入代码:

```
TESTINLINE.10:Inline($8b/$46/<i);(*MOV AX,I*)
```

```
CS:006 8B06      MOV   AX,[BP+06]
```

```
TESTINLINE.11:INline($46/<j);(*ADD AX,J*)
```

```
CS:0009 034604    add  ax,[DP+04]
```

```
TESTINLINE.12:inline($89/$46/$FE);(*MOV [BP-2],AX*);      CS:00C
8946FE      MOV   [bp-02],Ax
```

在调试时应注意比较嵌入代码和反汇编行的机器指令,看看是否为所希望的。过程的结束代码是把函数结果放入 AX 寄存器中,清理堆栈,返回到程序中的入口点:CS:000F 8B46FE

```
MOV   ax,[bp-02]
```

```
CS:0012 89EC      MOV   sp,bp
```

```
CS:0014 50        POP
```

```
CS:0015 C2040     ret   004
```

调试嵌入代码特别困难,因为它纯粹是由表示机器指令的数码组成。如果不是非常熟悉机器语言的数字代码,很难了解每条指令的确切含义。Turbo Debugger 把嵌入代码表示成汇编指令,逐条执行每一条指令,这样可以找到出问题的区域。Turbo Debugger 不仅适用于嵌入代码,而且同样适用于外部过程和 Pascal 代码。还可以用它学以用汇编语言编写程序,以及了解好的程序员是如何编程的。

## § 7.4 视频显示基本例程

有些程序使用得很普遍,几乎所有的程序都要用到这些程序,把好的过程和函数收集起来,需要时可以从其中取出应用是一种很好的方法。每个程序员都应该建立自己的过程和函数库。这样在编程时可以节省大量时间和精力。而且使用库中的过程和函数也容易调试和维护,较少出错。

本节和后面几节介绍一些常用的程序,可以把它们放入自己的库中,本节介绍几个与视频显示有关的程序,最后介绍一个产生声音的程序。

先介绍一个快速显示文本的程序。这个程序是用嵌入代码写的。显示文本的最快方法是将数据直接写入视频存贮区。下面的这个过程 Fastwrite 就是将字符串写到视频区。X 和 Y 是屏幕坐标,stype 表支色彩, M 表示单色, C 表示释色显示

```
Procedure FastWrite(x,y:Integer;
```

```
Var S:String;
```

```
fg,bg:Integer;
```

```
stype:Char);
```

```
Var
```

```
i,b:Byte;
```

```
Begin
```

If UpCase(Stype) = 'M' Then

Begin

b := (bg shl 4) + fg;

x := (x-1) \* 2 + ((Y-1) \* 160);

For i := 1 To Length(s) Do

Begin

Mem[\$B00:X] := Byte(s[i]);

Mem[\$b00:X+1] := b;

Inc(x,2);

End;

End

Else

Inline (\$50/	( * PUSH AX	*)
\$53/	( * PUSH BX	*)
\$51/	( * PUSH CX	*)
\$52/	( * PUSH DX	*)
\$1E/	( * PUSH DX	*)
\$06/	( * PUSH ES	*)
\$57/	( * PUSH DI	*)
\$56/	( * PUSH SI	*)
\$8B/\$5E/<X/	( * MOV BX,X	*)
\$8B/\$46/<Y/	( * MOV AX,Y	*)
\$4B/	( * DEC BX	*)
\$48/	( * DEC AX	*)
\$B9/\$50/\$00/	( * MOV CX,0050	*)
\$F7/\$E1/	( * MUL CX	*)
\$03/\$C3/	( * ADD AX,BX	*)
\$B8/\$02/\$00/	( * MOV CX,0002	*)
\$F7/\$E1/	( * MUL CX	*)
\$8B/\$F8/	( * MOV DI,AX	*)
\$8B/\$5E/<bg/	( * MOV BX,bg	*)
\$8B/\$46/<fg/	( * MOV AX,fg	*)

\$B9/\$04/\$00/	(* MOV CX,0004	*)
\$D3/\$E3/	(* SHL BX,CL	*)
\$03/\$D8/	(* ADD BX,AX	*)
\$86/\$DF/	(* XCHG BL,BH	*)
\$BA/\$DA/\$03/	(* MOV DX,03DA	*)
\$B8/\$00/\$B8/	(* MOV AX,B800	*)
\$8E/\$C0/	(* MOV ES,AX	*)
\$C5/\$76/<S/	(* LDS SI,s	*)
\$8A/\$0C/	(* MOV CL,[SI]	*)
\$80/\$F9/\$00/	(* CMP CL,00	*)
\$74/\$15/	(* JZ 2E06	*)
\$FC/	(* CLD	*)
\$46/	(* INC SI	*)
\$8A/\$1C/	(* MOV BL,[SI]	*)
\$EC/	(* IN AL,DX	*)
\$A8/\$01/	(* TEST AL,01	*)
\$75/\$FB/	(* JNZ 2DF5	*)
\$FA/	(* CLI	*)
\$EC/	(* IN AL,DX	*)
\$A8/\$01/	(* TEST AL,01	*)
\$74/\$FB/	(* JZ 2DF5	*)
\$8B/\$C3/	(* MOV AX,BX	*)
\$AB/	(* STOSW	*)
\$FB/	(* STI	*)
\$E2/\$EC/	(* LOOP 2DF2	*)
\$5E/	(* POP SI	*)
\$5F/	(* POP DI	*)
\$07/	(* POP ES	*)
\$1F/	(* POP DS	*)



```

$5A/          ( * POP    DX      * )
$59/          ( * POP    CX      * )
$5B/          ( * POP    BX      * )
$58/          ( * POP    AX      * )
$E9/$00/$00/  ( * JMP    2E11    * )
$8B/$E5/      ( * MOV    SP,BP   * )
$5D/          ( * MOV    BP      * )
$C2/$0E/$00); ( * RET    000E    * )

```

End;

这个程序首先判断是单色显示还是彩色显示，若是单色显示，程序走的分枝是用 Turbo Pascal 写的，这段程序把字符串直接送入以 B000:00 开始的单色显示存储区。若是彩色显示，其程序分枝是用嵌入代码编写的，它把字符串直接送入以 B800:00 开始的彩色显示区。

在许多与视频显示有关的程序中，都依赖计算机配备的是哪种显示适配器。在前面入 Fastwrite 就要首先知道是单显还是彩显。下面的这个程序 getscreentype 可以取得这个信息，Getscreentype 使用 BIOS 的 10h 中断，AH 为 0Fh，这个中断返回当前视频模式。如果 AL 中返回的代码为 7 时，表明当前的显示适配器为单色。下面是这个程序的列表：

```
procedure Getscreen—type (Var stype;Char);
```

```
Var
```

```
    Regs ; Registers;
```

```
Begin
```

```
    Regs.AH := $0F;
```

```
    Intr($10, Regs);
```

```
    If Regs.AL = 7 Then
```

```
        Stype := 'M'
```

```
    Else
```

```
        stype := 'C';
```

```
End;
```

下面的程序有三个过程，用于控制屏幕显示的光标的大小或关闭光标。它们使用 BIOS 的 10H 中断。第一个过程是关闭光标。这个过程首先使寄存器变量 Regs 中的 AH 为 01，CH 和 CL 都为 20H，然后调用 10H 中断：

```
Procedure Cursor—off;
```

```
Var
```

```
    Regs; Registers;
```

```
Begin
```

```
With Regs Do
```

```

Begin
  AH:= $ 01;
  CH:= $ 20;
  CL:= $ 20;
End
Intr( $ 10,Regs);
End;

```

使光标变小的过程如下。这个过程需求一个参数 stype, 它表示显示类型。当为彩色显示时 stype 为 'C'。当为单色显示时, Stype 为 'M'。然后据不同的显示方式, 设置不同的寄存然变量值, 然后调用10H 中断。

```

Procedute cursor-- Small(stype;char);
Var
  Regs;Registers;
Begin
  Case stype of
    'M':
      Begin
        With Regs Do
          Begin
            AH:= 1;
            CH:= 12;
            CL:= 13;
          End;
        End;
    'C':
      Begin
        With Regs Do
          Begin
            AH:= 1;
            CH:= 6;
            CL:= 7;
          End;
        End;
      End;
    Intr( $ 10,Regs);
  End;

```

使光标变大的过程与使光标变小的过程差不多, 不同的只是变量的值不一样。这时只要把上面过程中的 Case 分枝中的 CH 值改变一下, 在单色显示下 CH:=12换成 CH:=0;或在彩色显示方式下的 CH:=6; 换成 CH:=0;即可。

下面一个例子是在屏幕上指定行的中间显示一个字符串。这个过程与前面介绍的 Fast-

write 基本相同。不同的只是不必指定 X 光标，这个坐标将由过程计算，以使字符串在一行的中间显示。

```
Procedure Center(y:Integer;  
                S:string;  
                fg,bg:Integer;  
                stype:Char);  
  
Var  
    x: Integer;  
Begin  
    X:=40-(Length(s)Div2);  
    Fastwrite(x,y,fg,bg,stype);  
End;
```

下面介绍一个与显示无关，但很有意思的程序。它可以按给定的频率和时间发出声音。在这段程序中使用了几个 Turbo Pascal 命令。Sound 用于控制 PC 机的扬声器发声，它要求带一个表示频高低的参数。Sound 产生的声音一直持续到发出 Nosound 命令。Nosound 命令使声音停止。还要使用 Delay 控制声音的长短。Delay 要求带一个参数表示延迟的时间。单位是毫秒。下面这个过程 Beep 可以按指定时间和频率发声，它要带二个参数，Freq 用于确定声音频率的高低。Time 表示持续的时间。

```
Procedure Beep(Freq,Time:Integer);  
Begin  
    Sound(Freq);  
    Delay(Time);  
    Nosound;  
End;
```

## § 7.5 带缓冲字符串输入

虽然 Turbo Pascal 提供了输入过程 Read 和 Readln，但是它们的功能有限。例如使用这两个过程不能检测出是否按了某个功能键。在许多场合。这两个过程显得很不够用。为此可以使用下面两个过程对键盘控制做扩充。

Inkey 过程检测是否按了功能键。在 PC 机中按下一个功能键后，将产生一个扫描码和一个字符。Inkey 过程检测扫描码，如果扫描码存在，则将参数置为 TRUE。在 PC 机中，扫描码为 #0，要把功能键都记住比较困难，为此，应为 Inkey 设置一个全局枚举变量。这样就可以用句名字来引用功能键。并用一个全局变量 Key 存贮所按键的代码。例如：

```
Type  
    Keys=(Nullkey, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, carriagereturn, Tab, shift-  
        tab, Bksp, upArrow, DownArrow, RightArrow, LeftArrow, Deletekey, In-  
        sertkey, HomeKey, Esc, endkey, textkey, Numberkey, space, pgup, pgdn);  
  
Var  
    key:keys;
```

Inkey 还允许控制光标, 参数 BeginCursor 确定在等待按键时, 光标的形状, Endcursor 确定在按键接受后光标的形状。Inkey 共带四个参数。前面已经介绍的 Begin—Cursor 和 Endcursor 表示光标的形状。“S”表示小光标, “B”表示大光标, “O”表示关闭光标。参数 FunctionKey 为一个布尔变量, 当按下功级键时, 这个参数设置为 True, 否则设置为 False。参数 ch 放置所按的字符。当按了相应键后, 把枚举变量 keys 中相应的值放入 key 中。下面是 Inkey 过程的列表。

```

Procedure InKey(Var FunctionKey : Boolean;
                Var ch : Char;
                BeginCursor,
                EndCursor : Char);

```

```

Begin

```

```

    Case BeginCursor Of
        'B' : Cursor—Big(Stype);
        'S' : Cursor—Small(Stype);
        'O' : Cursor—Off(Stype);
    End;

```

```

    FunctionKey := False;
    ch := ReadKey;
    If (ch = #0) Then
        Begin
            FunctionKey := True;
            ch := ReadKey;
        End;

```

```

    If FunctionKey Then

```

```

        Case Ord(ch) of
            15: (* shift Tab *)   Key := Shiftab;
            72: (* up arrow *)    Key := UpArrow;
            80: (* down arrow *)  Key := DownArrow;
            82: (* insert key *)  key := InsertKey;
            75: (* left arrow *)  Key := LeftArrow;
            77: (* righth arrow *) Key := RightArrow;
            73: (* pge up *)      Key := PgUp;
            81: (* Pge down *)    Key := PgDn;
            71: (* home *)        Key := HomeKey;
            79: (* End *)         Key := enKey;
            83: (* delete *)      Key := DeleteKey;

```

```

      82: (* insert *)      Key := InsertKey;
      59: (* F1 *)          Key := F1;
      60: (* F2 *)          Key := F2;
      61: (* F3 *)          Key := F3;
      62: (* F4 *)          Key := F4;
      63: (* F5 *)          Key := F5;
      64: (* F6 *)          Key := F6;
      65: (* F7 *)          Key := F7;
      66: (* F8 *)          Key := F8;
      67: (* F9 *)          Key := F9;
      68: (* F10 *)         Key := F10;
      End
Else
      Case Ord(ch) of
      8: (* back Space *)    Key := Bksp;
      9: (* Tab key *)       Key := Tab;
      13: (* return *)      Key := CarriagerRetrun;
      27: (* estcape *)     Key := Esc;
      32: (* space bar *)   Key := Space;

      33..44, 47, 58..254;
      (* textKey *)         Key := TextKey;

      45..46, 48..57;
      (* number dey *)      Key := NumberKey;
      End;

      Case EnCursor Of
      'B' : Cursor—Big(Stype);
      'S' : Cursor—Small(Stype);
      'O' : Cursor—Off(Stype);
      End;

      End;

```

Inputstring shift 过程输入一个长字符串。这个字符串的长度可以比屏幕上所留的空白区更长。在输入过程中，可以用 BACKSPACE 和 DEL 键 删除字符，用光标控制键 RIGHTARROW 和 LEFTARROW 使光标前后移动，并且可以用 INS 键在插入和重写两种模式之间进行切换。

在这段程序中到用到几个标准 Turbo Pascal 过程。gotoXY(x,y)要求带两个整型参数 X, Y。这个函数的功能是将光标置于屏幕坐标(X,Y)处。COPY 函数要求三个参数，第一个是字

字符串 S, 第二个和第二个是整型。其基本形式为: COPY(S,P,L), 功能是返回字符串 S 中第 P 位开始的 L 个字符的子字符串。POS(SUBS,S) 函数要求二个字符串参数, 返回字符串 subs 在字符串 S 中的位置。Delete(S,P,L) 带三个参数, S 是字符串, D 和 L 为整型。它将删除字符串 S 中第 P 位开始的 L 个字符。

```

Procedure InprtStringShift(Var S : String;
                           WindowLengt,
                           MaxLength,
                           X,Y : Integer;
                           FT : Char;
                           BackgroundChar : Integer);

```

```

Var
  XX, i, j, P : Integer;
  ch : Char;
  InsertOn,
  SpecialKey : Boolean;
  offset : Integer;
  TempStr : String;

```

```

Procedure XY(x, y : Integer);

```

```

Var
  Xsmall : Integer;
Begin
  Repeat
    Xsmall := x - 80;
    If Xsmall > 0 then
      Begin
        Y := y + 1;
        X := Xsmall;
      End;
  Until Xsmall <= 0;
  GotoXY(x, y);
End;

```

```

( * * * * * )

```

```

Procedure SetString;

```

```

Var
  i : Integer;
Begin
  i := Length(s);

```

While s[i] = Char(BackgroundChar) do

    i := i-1;

s[0] := Char(i);

cursor--small(stype);

End;

( \* \* \* \* \* )

Begin

j := Length(s)+1;

For i := J To MaxLength do

    s[i] := Char(BackgroundChar);

s[0] := Char(MaxLength);

TempStr := Copy(s, 1, windowLength);

Fastwrite(x,y,tempStr, Yellow, Black, stype);

P := 1;

offset := 1;

InsertOn := True

Repeat

XX := X+(P-offset);

If (p-offset) = WindowLength The

    xx := xx-1;

XY(XX, Y)

If InsetOn Then

    InKey(SpecialKey, ch, 'S', 'O')

Else

    InKey(SpecialKey, ch, 'B', 'O');

If (FT = 'N') Then

    Begin

        If (Key = Textkey;

        End

Else If (ch = '-') and ((p > 1) Or (s[1] = '-')) Then

    Begin

        beep(100,250);

        key := NullKey;

    End

```

Else If (ch = '.') then
  Begin
    If Not(( Pos('.', s) = 0) Or (Pos('.', s) = p)) Then
      Begin
        beep(100,250);
        key := NullKey;
      End
    Else If (ch = '.') Then
      Begin
        If Not((Pos('.', s) Or (Pos('.', s) = p) Then
          Begin
            beep(100,250);
            key := NullKey;
          End
        Else If (Pos('.', s) = p) Then
          Delete(s, p, 1);
        End;
      End;
    End;
  Case key Of

```

```

    NumberKey,
    TextKey,
    Space ;
    Begin
      If (Length(s) = MaxLength) Then
        Begin
          If P = MaxLegnth Then
            Begin
              Delete(s, MaxLength, 1);
              s := s+ch;
              If p = windowLength+offset Then
                offset := offset+1;
              TempStr := Copy(s, offset, WindowLength);
              FastWrite(x,y,tempStr, Yellow, Black, stype);
            End
          Else
            Begin
              If InsertOn Then
                Begin
                  Delete(s, MaxLength, 1);

```



```

    Insert(ch, s, p);
    If p = WindowLength+offset Then
        offset := offset+1;
        If p < MaxLength Then
            p := p+1;
        TempStr := copy(s, offset, WindowLength);
        FastWrite(x,y,TempStr,Yellow,Black,stype);
    End
Else      (* overwrite *)
    Begin
        Delete(s, p, 1);
        Insert(ch, s, p);
        If p = WindowLength+offset Then
            offset := offset+1;
        If p < MaxLength Then
            p := p+1;
        TempStr := Copy(s, offset, WindowLength);
        FastWrite(x,y,TempStr,Yellow,Black,stype);
    End;
End;
Else
Begin
If InsertOn Then
    Begin
        Insert(ch, s, p);
    End
Else
    Begin
        Dlete(s, p, 1);
        Insert(ch, s, p);
    End;
    If p < MaxLength Then
        p := p+1;
        TempStr := Copy(s, offset, WindowLength);
        FastWrite(x,y,TempStr,Yellow,Black,stype);
    End
End

Bksp ;
    Begin

```

```

If p > 1 Then
  Begin
    p := p - 1;
    Delete(s, p, 1);
    s := s + Char(BackgroundChar);
    If offset > 1 Then
      offset := offset - 1;
    TempStr := Copy(s, offset, WindowLength);
    FastWrite(x, y, TempStr, Yellow, Black, stype);
    ch := ' ';
  End
Else
  Begin
    beep(100, 250);
    ch := ' ';
    p := 1;
  End
End

```

LeftArrow :

```

  Begin
    If P > 1 Then
      Begin
        p := p - 1;
        If p < offset Then
          Begin
            offset := offset - 1;
            TempStr := Copy(s, offset, WindowLength);
            Fastwrite(x, y, tempStr, Yellow, Black, stype);
          End;
        End
      End
    Else
      Begin
        SetString;
        Exit;
      End;
    End;
  End;

```

RightArrow :

```

  Begin

```

```

If (s[p] <> Char(BackgroundChar)) And (p < MaxLength)
  Then Begin
    P := p+1;
    If P = (WindowLength+offset) Then
      Begin
        offset := offset+1;
      TempStr := Copy(s, offset, WindowLength);
        FastWrite(x,y,TempStr,Yellow,Black,stype);
      End;
    End
  Else
    Begin
      SetString;
      Exit;
    End;
  End;
Deletekey :
  Begin
    Delete(s, p, 1);
    s := s+Char(BackgruondChar);
    If ((Length(s)+1-offset) >= WindowLength) Then
      Begin
        TempStr := Copy(s, offset, WindowLength);
        FastWrite(x,y,TempStr,Yellow,Black,stype);
      End
    Else
      Begin
        TempStr := copy(s, offset, WindowLength);
        FastWrite(x,y,tempStr,Yellow,Balack,stype);
      End;
    End;
InsertKey :
  Begin
    If InsertOn Then
      InsertOn := False
    Else
      InsertOn := True;
    End;

```

```
Else If Not(key In [CarriageReturn, UpArrow, DownArrow, PgDn, PgUp, Nullkey, Esc,
    Tab, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10]) Then beep(100,
    250);
```

```
End;
```

```
Until (key In [CarriageReturn, UpArrow, DownArrow, PgUp, PgDn, Esc, Tab, F1,
    F2, F3, F4, F5, F6, F7, F8, F9, F10]);
```

```
SetString;
```

```
End;
```

## § 7.6 大字符串的处理

在 Turbo Pascal 中字符串的长度限制为255 个字符,一般来讲,对大多数应用是足够的。但是有时可能需要更长的字符串。下面的一组过程表明如何处理大字符串,包括大字符串的初始化,大字符串的连接等等。这种长字符串可以定义为一个记录类型,包含两个域。一个整型域记录串的长度和一个字符数组。例如

```
const
    MaxBigstrlen=1000;
type
    Bigstring=Record
        Length;integer;
        Ch:Array[1..MaxBigstrlen] of Char;
    End;
```

这里定义的字符串 Bigstring 最大可以有1000个字符。通过改变 MaxBigstrlen 的值,可以很容易扩大字符串的容量。最大可以有32767个字符。下面的几个过程和函数模拟 Turbo Pascal 的标准字符串的过程,使用相同的语法和名字,但是在名字前加上"Big"。例如与 Instert 等价的 BigInsert。

下面分别介绍:

SetBigstring—这个过程用参数 S 中给出的值初始化一个大字符串 St1。

```
Procedure setbigstring(var st1;Bigstring; S: String);
```

```
Var
```

```
    i;integer;
```

```
Begin
```

```
For i:=1 to Length(S) Do
```

```
    St1.ch[i]:=S[i];
```

```
St1.Length:=Length(s);
```

```
End;
```

Bigconcat—这个过程模拟 Turbo Pascal 中的 concat 命令。对于大字符串,不能用操作符

十、Bigconcat 把大字符串 St2接到 St1后面。

```
Procodure Bigconcat (Var st1; Bigstring;  
                      St2; Bigstring);
```

```
Var
```

```
  i; Integer;
```

```
Begin
```

```
  Mov (St2. ch[1], st1. ch[st1. length+1], st2. length);
```

```
  st1. Length := st1. Length + st2. Length;
```

```
End;
```

这个过程中用到一个标准 Turbo Pascal 过程 Mov。它要求三个参数，第三个参数是整型数。Mov 的一般形式为：

```
Mov (var v1, v2, i; Integer);
```

把 v1 中的 i 个字节复制到 v2 中。

BigInsert—这个过程在目标大字符串中的指定字符开始处插入另一个大字符串。这个过程要求三个参数：第一和第二个是大字符串，第三个是整型变量，下面是 BigInsert 过程：

```
Procodure BigInstert (Var St1, St2; BigInsert; P; Integer);
```

```
Var
```

```
  St3; Bigstring;
```

```
  i, j; Ineger;
```

```
Begin
```

```
  Mov (St2. ch[1], st3. ch[1], p-1);
```

```
  Mov (St1. Ch[1], St3. ch[p], st1. Length);
```

```
  Mov (St2. ch[p], St3. ch[p+st1. Length], (st2. Length-P)+1);
```

```
  St3. Length := st1. Lenggth + st2. Length;
```

```
  If st3. Length > MaxBigstrLen;
```

```
  Mov (st3. Length, St2. Length, St3. Length+2);
```

```
end;
```

这个过程把 st1 插入到 st2 的 P 位中。如果得到的大字符串大于大字符串的最大长度，则截掉后面的字符。

BigDelete—这个过程从大字符串中删掉若干字符。

```
procedure BigDelete (Var St1; Bigstring; P, len ; Integer);
```

```
Var
```

```
  St2; Bigstring)
```

```
Begin
```

```
  Mov (St1. ch[1], St2. ch[1], p-P);
```

```
  Mov (St1. ch[p+Len], St2[p], (st1. Length - (p+Len))+1);
```

```
  St2. Leagth := St1. Length - Len;
```

```
  Mov (St2. Length, St1. Length, St2. Length+2);
```

```
End;
```

这个过程删除大字符串 St1 中第 P 个字符开始的 Len 个字符。

BigPos—这个函数返回一个大字符串在另一个大字符串中的位置。如果大字符串不在另一个大字符串中，则返回0。

```
Function BigPos(Var St1,St2:Bigstring):Integer;
```

```
Var
```

```
    found:Boolean;
```

```
    i, j, StopFlag:Integer)
```

```
Begin
```

```
Stopflag:=(st2.Length-st1.Length)+1;
```

```
For i:=1 To StopFlag Do
```

```
    begin
```

```
        found:=True;
```

```
        j:=1;
```

```
        Repeat
```

```
            If St2.ch[i+j-1]<>st1.ch[j] Then
```

```
                found:=False;
```

```
            i:=j+1;
```

```
            Until(Not found) or (j=st1.langlin);
```

```
        If found Then
```

```
            Begin
```

```
                BigPos:=i;
```

```
            Exit;
```

```
            End;
```

```
        End;
```

```
Bigpos:=0;
```

```
End;
```

这个函数返回大字符串 St1 在大字符串 St2 中的位置。如果 ST1 不在 ST2 中。则返回0。

Biglength—这个函数返回大字符串的长度。

```
Function BiLength(var st:Bigstring):Integer;
```

```
Begin
```

```
Biglenth:=st.Length;
```

```
end;
```

Bigcopy—这个过程用于从一个大字符串中取出若干字符。Turbo Pascal 的 copy 函数不能直接用于大字符串。因为大字符串是记录型，而函数不能说明为记录型。

```
Procodure Bigcopy(Var St1,ST2:Bigstring; p, Len:Integer);
```

```
Begin
```

```
Mov(St1,ch[p],St2,ch[1],Len);
```

```
St2.Length:=Len;
```

```
End;
```

这个过程把 St1 中第 P 个字符开始的 Len 个字符放到 st2 中。

## § 7.7 算术函数

Turbo Pascal 提供的函数和过程可以解决大部分与数值有关的问题。但是与分数有关的操作却没有相应的功能。例如将字符串中的数转换成相应实数，或将小数转换为相应的字符串。下面的两个过程实现了这两个功能。

第一个函数是 Real—To—Frac，要求有两个参数，r 是要转换的数，d 是要转换成分数的分母，Real—To—Frac 函数返回一个字符串，分成整数部分和分数部分。这两部分之间用一个空格分开。

```
Function Real—To—Frac(r : Real; d : Integer) : String;
```

```
Var
```

```
    is, ns, ds, sl, s2 : String[20];
```

```
    r1, r2, i, f : Real;
```

```
    code, p, n : Integer;
```

```
Begin
```

```
    If r = 0 Then
```

```
        Begin
```

```
            Real—to—Frac := '0';
```

```
            Exit;
```

```
        End;
```

```
    is := '0';
```

```
    ds := '0';
```

```
    ns := '0';
```

```
    str(0:8, s2);
```

```
    p := Pos('.', s2);
```

```
    If p > 0 Then
```

```
        si := Copy(s2, 1, p-1);
```

```
    Delete(s2, 1, p-1);
```

```
    Val(s1, i, code);
```

```
    Str(i, 0:0, is);
```

```
    Val(s2, f, code);
```

```
    If f > 0.0 Then
```

```
        Begin
```

```
            n := 0;
```

```
            Repeat
```

```

    n := n + 1;
    r1 := n/d;
    Until r1 >= f;
If (r1 - f) > (1.0/(d * 2.0)) Then
    n := n - 1;

While (Not Odd(n)) And (n > 0) Do
    Begin
        n := n DIV 2;
        d := d DIV 2;
    End;
Str(n;0, ns);
Str(d;0, ds);
End;

If (ns = '1') And (ds = '1') Then
    Begin
        ns := '0';
        Val(is,r1,code);
        r1 := r1 + 1;
        Str(r1;0;0, is);
    End;

If (is = '0') And (ns = '0') Then
    Real—To—Frac := '0'
Else If ns = '0' Then
    Real—To—Frac := is
Else If is = '0' Then
    Begin
        If (ns = '1') And (ds = '1') Then
            Real—To—Frac := '1'
        Else
            Real—To—Frac := ns + '/' + ds;
    End
Else
    Real—to—Frac := is + ' ' + '/' + ds
End;

```

这个过程中用到两个标准 Turbo Pascal 过程：str 和 val，Str 是将一个实数转换成一个字符串。Var 是将一个字符串转换成数值。

Real—To—Frac 过程首先判断 r 是否为 0。如果 r=0，则返回 0。若 r 不为 0，则将 r 分成



整数部分和小数部分。将小数部分转换成分母为 d 的分数。

第二个过程是 Frac—To—Real，这是一个实数型函数。它将一个字符串转换成实数，字符串由整数部分，空格，分子，斜线和分母组成。可以只有整数部分，或只有小数部分，也可以两部分都有。如果整数部分和小数部分都有，之间由空格分开。Frac—To—Real 要求两个参数，Frac 是一个字符串，Code 是错误指示代码。若 Code 为 0，表示转换正确。否则表示有错误。

这个过程首先将字符串分为整数部分和分数部分，然后分别转换为数值。

## § 7.8 文件加密

经常有必要保护自己的文件，数据和程序，防止无关的人员使用。保护方法有各种各样，将文件加密是一种行之有效的方法。下面的两个过程分别用于文件的加密和脱密。

Encode 是加密过程，它要求输入文件名和通行字。Encode 根据通行字对文件进行加密。首先用通行字产生两个种子数，用这两个种子数控制加密。Encode 将这两个种子数放在被加密文件中，任何人没有正确的口令将不能对加密文件进行脱密。口令最长可有六个字符。

加过密的文件的前六个字节为 'LOCKED'。Encode 在加密前先检查文件的前六个字母，若为 'LOCKED'，程序将显示 File already locked，并终止程序的执行。这样可以防止对同一个文件进行二次加密。

为防止无关者用特殊程序浏览磁盘上文件的发现文件的内容。Encode 先用二进制的对原文件重写一遍，再删去。下面是 Encode 过程的列表：

```
Program encode;
```

```
Const
```

```
    MaxBuf = 30000;
```

```
Var
```

```
    password : String[6];
```

```
    seed1,
```

```
    seed2 : Byte;
```

```
    source,
```

```
    dest : File;
```

```
    buffer : Array [1..MaxBuf] Of Byte;
```

```
    BytesRead : Real;
```

```
    i : Integer;
```

```
( * * * * * )
```

```
Procedure OpenFiles;
```

```
Const
```

```
    s : Array [1..6] Of Char = ('O','O','C','K','E','D');
```

```
Begin
```

```
    Assign(source,ParamStr(1));
```

```
( * $ I - * )
```

```

Reset(source,1);
(* $U+ *)
If IOresult <> 0 Then
  Begin
  WriteLn('File not found. ');
  Halt;
  End;

```

```

BlockRead(source,buffer,6);
If ((buffer[1] = ord('L')) and
    (buffer[2] = ord('O')) And
    (buffer[3] = ord('C')) And
    (buffer[4] = ord('K')) And
    (buffer[5] = ord('E')) And
    (buffer[6] = ord('D')) And
    Begin
    WriteLn('File already locked. ');
    Halt;
    End;

```

```

Reset(source,1);
Assign(dest,'$ $ $ $ $ $ $ $ ');
Rewrite(dest,1);
BlockWrite(dest,s,6);
BlockWrite(dest,seed1,1);
BlockWrite(dest,seed2,1);
End;

```

( \* \* \* \* \* )

```

Procedure Getseed;
Var
  i,j ; Integer;
Begin
  seed1 := 0;
  seed2 := 0;
  password := ParamStr(2);

  j := Length(password);
  For i := 1 To Length(password) Do

```

```

    Begin
    seed1 := seed1 + (Ord(password[i] * i);
    seed2 := seed2 + (Ord(password[i] * j)
    j := j - 1;
    End;
End;

```

( \* \* \* \* \* )

```

Procedure EncodeFile;
Var  i1,i2, : Byte;
    rr : Integer;
Begin
i1 := seed1;
i2 := seed2;
BytesRead := 0;
BlockRead(source,buffer,MaxBuf,rr);
BytesRead := BytesRead + rr;
While rr > 0 Do
    Begin
    For i := 1 To rr Do
        Begin
        i1 := i1 - i;
        i2 := i2 + i;
        If odd(i) Then
            buffer[i] := buffer[i] - i1
        Else
            buffer[i] := buffer[i] + i2;
        End;
        BlockWrite(dest,buffer,rr);
        BlockRead(source,buffer,MaxBuf,rr);
        BytesRead := BytesRead + rr;
        End;
    End;
End;

```

( \* \* \* \* \* )

```

Procedure CloseFiles;
Var
    i := Integer;

```

```

Begin
Rewrite(source,1);
Fillchar(buffer,MaxBuf,0);
While BytesRead > 0 Do
  Begin
    If BytesRead > MaxBuf Then
      BlockWrite(source,buffer,MaxBuf)
    Else
      Begin
        i := Trunc(BytesRead);
        BlockWrite(source,buffer,i)
      End;
    BytesRead := BytesRead - MaxBuf;
  End;
Close(source);
Close(dest);
Erase(source);
Rename(dest,ParamStr(1));
End;

```

( \* \* \* \* \* )

```

Begin
If Paramcount <> 2 Then
  Begin
    WriteLn('Syntax: ENCODEIT Filename password');
    Halt;
  End;
Getseed;
OpenFiles;
EncodeFiles;
CloseFiles;
End.

```

这个过程启动时使用下面格式:

Encode<文件名><通行字>

如果格式不正确,则程序显示 'SYNLAX;ENCODEIT Filename Password' 等停止程序的执行。如果文件名不存在,则显示 'File Notfound' 考停止程序的执行。encode 先用通行字产生两个种子数,希望打开文件。对文件进行加密。

加密后的文件于关从没不能看懂其内容。若自己要恢复文件的原来形式可用 DECODE 对文件进行脱密。decode 的格式与 Encode 的一样:

Decode 主件名<><通行字>

如果格式不正确,则显示 'Sgnlax;dECODEIT Filename Password' 并终止程序的执行.如果文件不存在,则显示 'file not fond' 停止程序的执行.Decode 先用检查文件中是否有 'LOCKED',如果没有则显示 'File not lecked' 停止执行程序.然后用通行字产生两个种子数,与文件中的种子数进行比较.如果不正确则显示 'Wrong Password' 停止执行程序.如果都正确,则对文件脱密.

Program Decode;

Const

MaxBuf = 30000;

Var

    passord : String[6];

    source,

    dest : File; buffer : Array [1..MaxBuf] Of Byte;

    BytesRead : Real;

    seed1,

    seed1x,

    seed2,

    seed2x : Byte;

    i : Integer;

( \* \* \* \* \* )

Procedure OpenFiles;

Const

    s : array [1..6] Of Char = ('L','O','C','K','E','D');

Begin

    Assign(source,ParamStr(1));

    (\* \$I- \*)

    Reset(source,1);

    (\* \$I+ \*)

    If IOresult <> 0 Then

        Begin

            WriteLn('File not found.');

            Halt;

        End;

    BlockRead(source,buffer,6);

    If Not ((buffer[1] = ord('L') And

        (buffer[2] = ord('O')) And

        (buffer[3] = ord('C')) And

        (buffer[4] = ord('K')) And

```

        (buffer[5] = ord('E')) And
        (buffer[6] = ord('D')) And
        Begin
            WriteLn('File already locked. ');
            Halt;
            End;

BlockRead(source, seed1x, 1);
blockRead(source, seed2x, 1);

If ((seed1 <> seed1x) Or (seed2 <> seed2x)) Then
    Begin
        WriteLn('Wrong password. ');
        Halt;
        End;

Assign(dest, '$ $ $ $ $ . $ $ ');
Rewrite(dest, 1);
End;

( * * * * * )

Procedure Getseed;
Var
    i, j : Integer;
Begin
    seed1 := 0;
    seed2 := 0;
    password := ParamStr(2);

    j := Length(password);
    For i := 1 To Length(password) Do
        Begin
            seed1 := seed1 + (Ord(password[i]) * i);
            seed2 := seed2 + (Ord(password[i]) * j);
            j := j - 1;
        End;
    End;

( * * * * * )

```

```

Procedure DcodeFile;
Var  i1,i2, : Byte;
      rr : Integer;
Begin
i1 := seed1;
i2 := seed2;
BytesRead := 0;
BlockRead(source,buffer,MaxBuf,rr);
BytesRead := BytesRead + rr;
While rr > 0 Do
  Begin
    For i := 1 To rr Do
      Begin
        i1 := i1 - i;
        i2 := i2 + i;
        If odd(i) Then
          buffer[i] := buffer[i] - i1
        Else
          buffer[i] := buffer[i] + i2;
      End;
    BlockWrite(dest,buffer,rr);
    BlockRead(source,buffer,MaxBuf,rr);
    BytesRead := BytesRead + rr;
  End;
End;

```

( \* \* \* \* \* )

```

Procedure CloseFiles;
Var
  i := Integer;
Begin
Rewrite(source,1);
Fillchar(buffer,MaxBuf,0);
While BytesRead > 0 Do
  Begin
    If BytesRead > MaxBuf Then
      BlockWrite(source,buffer,MaxBuf)
    Else
      Begin

```

```

        i := Trunc(BytesRead);
        BlockWrite(source,buffer,i)
    End;
    BytesRead := BytesRead - MaxBuf;
    End;
Close(source);
Close(dest);
Erase(source);
Rename(dest,ParamStr(1));
End;

( * * * * * )

Begin
If Paramcount <> 2 Then
    Begin
        WriteLn('Syntax: ENCODEIT Filename password');
        Halt;
    End;
Getseed;
OpenFiles;
DecodeFile;
CloseFiles;
End.

```



## 第八章 Turbo Pascal 工具箱

工具箱是 Turbo Pascal 提供的一组软件,用于完成某种特定工作。这里介绍四个工具箱:数据库工具箱,图形工具箱,编辑工具箱和数值方法工具箱。工具箱提供的工具可以有效地解决相应的问题,可以大大省略编程工作。

### § 8.1 数据库工具箱

这个工具箱主要有两部分:数据库过程和排序过程。数据库管理是微型计算机的主要应用之一。数据库的主要优点是具有索引功能。利用索引在数据库中查找数据要比顺序查找快得多。

Turbo Pascal 的数据库工具箱包括下列文件:

TACCESS. PAS	数据库说明和子程序
TAHIGH. PAS	高级数据库子程序
SORT. PAS	用于小于32767项的排序子程序
LSORT. PAS	用于大于32767项的排序子程序

这些文件都是可用于任何程序的单元。

例如在 DOSORT 程序中使用 SORT。可如下说明

```
Program Dosort;
```

```
Use SORT;
```

TACCESS 单元中包含创建数据库和索引文件所需的数据类型。数据库文件的数据类型定义如下:

```
DataFile=Record
    F : File
    FirstFree,
    NumberFree,
    Intl : longInt;
    Itemsize : Word;
    NumRec : LongInt;
End;
```

其中 NumRec 是总记录数, NumberFree 是删除的记录个数, FirstFree 是最后删除的记录, Itemsize 是记录的大小。Intl 为索引子程序使用。所有这些信息都放在数据库的第一个记录中。打开数据库时, TACCESS 首先读出这个记录启动数据库。

数据库工具箱提供了管理索引的子程序,每当向数据库增加一个记录时,也在索引中增

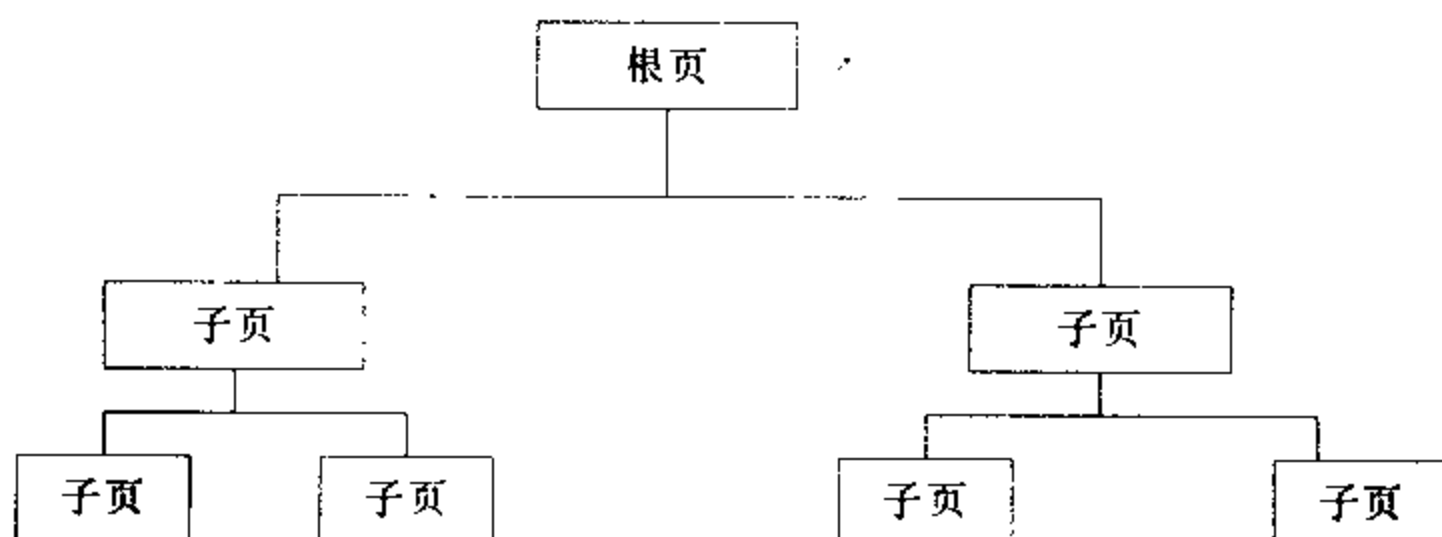
加一个关键字。删除记录时，也删除索引中的一个关键字。在修改记录时，如果改变了关键字，则索引也改变相应的关键字。可以指定是否允许重复关键字。

在数据库工具箱使用之前，要定义一些常量，控制文件请求的内存和盘空间的大小。

MaxDatasize—这个值决定数据库记录缓冲区的大小，这个值在8 到65535之间。这个值必须大于数据库中任何一个记录。

MaxKeyLen—当索引关键字是字符串时，这个值决定索引文件中允许的最大字符串长度。

MaxHeight—决定 B+ 树的最大高度。索引文件由 B+ 树结构组成，其基本构成块是页。其结构如下：



Order—决定页的大小。页必须为偶数，这个常数的2 倍等于页的大小。

Pagesize—决定索引中每页关键字的数目。这个值最小为4，最大为254。一般地这个值在30到50 之间。

Pagestacksize—决定存放于 RAM 中的索引页的数量。这个值大时，RAM 中存放的页也多，因而增加索引查找的速度。但是若太大，可能没有足够的内存来运行程序。建议一般应设置 Pagestacksize 值为16 到 32 之间。

上面这些值存放在 TACCESS. DEF 的文件中。可以自己创建这个文件，也可以用 TABUILD. EXE 程序帮助建立 TACCESS. DEF 文件。下面举例说明。在文本文件 DATA. TYP 中有下列数据说明：

Type

Datakey=String[20];

DataRec=Record

Name;String[30];

Age;Word;

Income;Real;

End

MaxDatatype=DataRec;

Maxkeytype=datakey;

这时，在 DOS 提示符下键入：

## TABUILD/W+DATA.TYP

TABUILD 计算数据和索引关键字的大小, 计算内存需求, 盘空间使用和其它重要信息, 显示在一个工作图表中。用户可以改变其中的三个值: 数据库中的记录个数, 每个索引页中关键字个数和存放在 RAM 中的页个数。下面是工作图表的例子:

数据库记录页大小 页数目 RAM 每次查找比较次数在 RAM 中的查找比例

1.000	64	5 9.305	1.78	66.72%
10.000	64	5 9.305	2.38	45.53%
1.000	20	5 2.925	2.55	49.66%
1.000	64	1018.610	1.78	80.06%
10.000	30	2017.500	3.70	49.88%

利用 TABUILD 提供的这个工作图表, 可以确定每个索引页中关键字的数目(页大小), 和存放在 RAM 中索引页的数目(页数目)的值。选择好后按 F2 键可自动创建 TACCESS.DEF 文件, 编译 TACCESS 单元。Maxdatarecsize 和 Maxkeylen 的值由 DATA.TYP 中的定义确定; Pagesize 和 Pagestacksize 是在 TABUILD 工作图表中选定的值; Order 和 Maxheight 由计算得到。使用 TABUILD 可以大大简化创建一个适合特定应用的数据库的过程。

下面这段程序说明了数据库工具箱常数:

Program sampledec;

Uses TACCESS;

Var

Dat=F;DataFile;

IdxF;IndexFile;

DatRef;LongInt;

其中 DatF 是数据库文件, IdxF 是索引文件, DatRef 是保存记录号的变量。记录号把数据库与索引连接在一起。当在索引中寻找一个关键字时, 若找到匹配的记录, 则索引程序返回其记录号。用这个记录号可从数据库中找到适当的记录。

Turbo Pascal 数据库工具箱中应用过程和函数, 可以分为两部分: 低级命令和高级命令。下面先介绍低级命令:

addkey

语法: Procedure addkey(Var IdxF;IndexFile; Var DataRef;LongInt; Var Key);

功能: Addkey 把关键字和对应的记录号 DatRef 加到索引文件 IdxF 中。

addrec

语法: Procedure addRec(Var DatF;DataFile; Var DatRef;LongInt; Var Buffer);

功能: 将 Buffer 中的记录加到数据库文件 DatF 中, 并返回记录号 DataRef。

ClearKey

语法: Procedure ClearKey(Var IdxF;IndexFile);

功能: 将索引文件的指针指向第一项。

#### CloseIndex

语法: Procedure CloseIndex(Var IdxF;IndexFile);

功能: 关闭一个打开的索引文件。

#### DeleteKey

语法: Procedure DeleteKey(Var IdxF;IndexFile; Var DatRef ;LongInt; Var Key)

功能: 从一个索引文件中删除一个关键字和其对应的记录号。

#### DeleteRec

语法: Procedure DeletRec(Var DatF;DataFile; Datref; LongInt);

功能: DeleteRec 删除一个记录的内容, 并将记录号加到删除记录表中

#### EraseFile

语法: Procedure EraseFile(Var DatF;DataFile);

功能: 删除一个数据库文件

#### EraseIndex

语法: Procedure EraseIndex(Var IdxF;IndexFile);

功能: 删除一个索引文件。

#### Filelen

语法: Function Filelen(Var DatF;DataFile);LongInt;

功能: 返回用记录数目表示的文件大小。包括活动记录, 删除记录和由数据库工具箱保留的记录。

#### FindKey

语法: Procedure FindKey(VAr IdxF;IndexFile; Var DatRef ; LongInt; Var Key);

功能: 查找等返回 与 Key 匹配的索引。如果找到, 返回对应的记录号, 并将 OK 设置为 TRUE。若未找到, OK 设置为 False, 并返回索引文件中的最后一个关键字。

#### FlushFile

语法: Procedure FlushFile(Var DatF;DataFile);

功能: 将 RAM 缓冲区中的数据写到磁盘上。

#### FlushIndex

语法: Procedure FlushIndex(Var IdxF;IndexFile);

功能: 将 RAM 缓冲区中的索引写到磁盘上去。

#### GetRec

语法: Procedure GetRec(Var DatF;DataFile; DatRef; LongInt; Var Buffer);

功能: 从数据文件 DatF 中检索一个记录号 DatRef, 将其内容放在 Buffer 中。GetRec 不影响变量 OK, 出现的任何错误由 Talockeek 处理。

#### MakeFile

语法: Procedure MakeFile(Var DatF;DatFile;File—Name; String; RecLen;Integer);

功能: 建立一个名为 FileName 的新数据文件, 其记录长度为 RecLen。

#### MakeIndex

语法: Procedure MakeIndex(Var IdxF;IndexFile;FileName; String; KeyLen, Status;Integer);

功能: 建立一个名为 FileName 的索引文件 IdxF, 其关键字允许的最大长度为 KeyLen。

Status 决定是否允许重复关键字。如果 Status 为0, 不允许重复关键字; 如果为1, 允许重复关键字。

#### NextKey

语法: Procedure NextKey(Var IdxF:IndexFile; Var DatRef: Integer; Var Key);

功能: 将索引文件的指针指向下一项, 并返回该项的关键字和记录号。如果文件指针已指向最后一项, 调用 NextKey 后, 指针将指向第一项。

#### OpenFile

语法: Procedure OpenFile(Var DatF:DataFile; FileName: String; RecLen:Integer);

功能: 打开一个已存在的数据库文件。RecLen 的值必须等于创建时使用的值。若执行成功, OK 被置为 TRUE。

语法: Procedure OpenIndex(Var IdxF:IndexFile; FileName: String; KeyLen, Status:Integer);

功能: 打开一个已存在的索引文件, KeyLen 必须等于建立索引文件时使用的值。

#### PrevKey

语法: Procedure PrevKey(Var IdxF:IndexFile; Var DatRef:LongInt; Var Key);

功能: 将索引文件的指针指向前一项, 并返回该项的关键字和记录号。如果在调用 ClearKey 之后调用 prevKey, 指针将指向最后一项。

#### putrec

语法: Procedure PutRec(Var DatF:DataFile; DatRef: LongInt; Varbuffer);

功能: 用 Buffer 中的内容替换数据库文件中的记录。DatRef 须大于1, 小于 FileLen。

#### SearchKey

语法: Procedure SearchKey(Var IdxF:IndexFile; Var DatRef:Integer; Var Key);

功能: 在索引文件中寻找大于或等于 Key 的第一个关键字。若找到, 则返回其关键字和记录号, 并将 OK 置为 TRUE。

#### UsedRecs

语法: Function UsedRecs(Var DatF:DataFile):LongInt;

功能: 返回数据库文件中活动记录的个数。

数据库工具箱的功能非常强大, 但用它维护索引文件时, 比较难于使用, 为此, 提供了一系列高级例程使之大大简化。这些高级例程在 TAHIGH 单元中。它将数据库限制于一个索引。这对大多数应用没有多大问题, 但大大简化了修改数据库索引的工作。

在 TAHIGH 单元中定义了一个数据类型 Dataset。

#### Type

Dataset=Record

Data;DataFile;

Index;IndexFile;

End;

下列是高级例程:

#### Taclose

语法: Procedure Taclose(Var Dataset:Dataset);

功能: 关闭 Dataset 数据库和索引文件。

TaCreate

语法: Procedure TaCreate (Var Datset; Dataset; DatFName; string; RecordLen; Integer;  
IndexFName; String; KeyLen; Integer);

功能: 建立一个名为 DatFName 的数据库文件和一个名为 IndexFName 的索引文件。记录长度由 RecordLen 决定, 索引关键字的长度由 KeyLen 决定。

TaDelete

语法: Procedure TaDelete (Var Datsat; Dataset; Var Key);

功能: 删除由 Key 标识的数据库记录和索引项。

TaErase

语法: Procedure TaErase (Var Datset; Dataset);

功能: 删除 Datset 标明的数据库文件和索引文件。

TaFlush

语法: Procedure TaFlush (Var Datset; Dataset);

功能: 将 RAM 缓冲区中的数据和索引记到数据库文件和索引文件中。

TaInsert

语法: Procedure TaInsert (Var Datset; Dataset; Var DataRec, Key);

功能: 在数据库中增加一个记录, 并在索引中增加一个关键字。

TaNext

语法: Procedure TaNaxt (Var Datset; Dataset; Var DataRec, key);

功能: 从数据库中检索对应索引中下一个关键字的记录。

TaOpen

语法: Procedure TaOPen (Var Datset; Dataset; DatFName;  
String; RecordLen; Integer; IndexFName  
;String; KegLen; Integer);

功能: 打开一个数据库及其索引文件。其记录长度为 RecordLen, 关键字长度为 KeyLen。

Taprev

语法: Procedure Taprev (Var Dataset; Var DataRec, Key);

功能: 从数据库中检索对应索引中前一个关键字的记录。

TaRead

语法: Procedure TaRead (Var Datset; Dataset; Var DataRec, Key; FindExact; Boolean);

功能: 从数据库中检索对应于 Key 的记录。如果 FindExact 为 TRUE, 则只检索与关键字 Key 精确匹配的记录。如果为 FALSE, 则关键字的前面部分匹配的记录即检出。

TaREset

语法: Procedure TaReset (Var Datset; Dataset);

功能: 将文件指针指向文件头。

TaUpdate

语法: Procedure TaUpdate (Var Datset; Dataset; Var Date  
Rec, Key);

功能: 用 DataREc 中的数据修改数据记录。

TaWrite

语法: procedure TaWrite(Var Dataset; Dataset; Var DataRec, Key);

功能: 向数据库写一个记录。

数据库工具箱的另一部分是排序程序。它有两个单元: SORT 单元和 LSORT 单元。这两个单元的功能相似, 只是 SORT 用于 32767 个以下的记录排序, LSORT 用于多于 32767 个记录的排序。尽管 LSORT 功能更强。但对少量数据进行排序时没有 SORT 的效率。

Turbo Sort 程序读进无序记录, 输出已排好的记录。把记录传给排序程序, 排序程序进行排序, 再将排好序的记录输出来。为此, 必须编写三个简单的过程: Inp, Less 和 Outp。

Inp 用于把记录传给 TurboSort 程序。下面是一个 Inp 过程的例子:

```
{ $F+ }
Procedure Inp;
Begin
Assign(InFile, 'TEMP. DAT'); Reset(InFile);
While Not Eof(InFile) Do
Begin
Read(InFile, DataRec);
WriteLn(DataRec);
SortRelease(DataRec);
End;
Close(InFile);
End;
{ $F- }
```

这个过程从数据文件 TEMP. DAT 中读出记录, 传给 TurboSort。Inp 过程结束后, 前进到 Less 的排序过程。

Less 函数用来定义 TurboSort 的排序方法。下面是一个简单 Less 函数的例子:

```
{ $F+ }
Function Less(Var x, y): Boolean;
Var
Rec1: DataREcType Absolute x;
Rec2: DataRectype Absolute y;
Begin
Less := Rec1 < rec2;
End;
{ $F- }
```

在这个函数中说明了两个用于排序的记录, Rec1 为 x 的绝对值, Rec2 为 y 的绝对值。在函数体中比较 Rec1 和 Rec2, 并将结果赋给函数 Less。TurboSort 调用 Less, 直到所有的数据项都已排好, 然后调用 Outp 过程。

Outp 使用 SortReturn 命令, 按排序顺序输出数据。用 SortEOS 来控制循环, 当所有数据项都输出完后, SortEOS 返回 TRUE。下面是一个 Outp 的例子:

```
{ $F+ }
Procedure Outp;
```

```

Begin
ClrScr;
  WriteLn;
  Repeat
    SortReturn(DataRec);
    WriteLn(DataREc);
  Until SorTEOS;
End;
{ $F- }

```

一旦定义了 Inp, Less 和 Outp 之后, 可以调用 turboSort 来排序了。TurboSort 要求一个指示数据到大小的参籽下面是一个排序的例子:

```
Res:=TurboSort(Size of(DataRec), @Inp, @Less, @vtp);
```

Res 赋以 TurboSort 的返回值。这个值是 TurboSort 的状态码, 其意义如下表:

代码	含 义	解决的方法
0	排序完成。	
3	没有足够的 RAM。	通过覆盖或动态变量决少程序内存的使用
8	排序数据项太短 *。	增加数据项的大小。
9	传给 TurboSort 的数据多于 MaxInt * *。	将文件分成若干小块, 分别排序, 然后合并各个文件。
10	写错误。磁盘已满或损坏。	检查磁盘的自由空间和可能的损坏。
11	读错误。	检查磁盘是否损坏。
12	文件创建错误	检查目录是部已满, 或试图访问一个不存在的目录。

\* 排序的数据项至少应为二字节。

\* \* Lsort 没有此限制。

## § 8.2 图形工具箱

图形工具箱有用户所需的大多数程序。用它可以管理屏幕, 窗口, 作图等等。下面介绍图形工具箱的命令和过程。

**Bezier**

语法: Procedure Bezier(A: PhtArray; N: Integer; Var B: PhtArray; M: Integer);

功能: 用平滑曲线将图上各点连接起来, A 是 X, Y 点的一个数组, N 是点的个数, B 是由 Bezier 计算出点 X, Y 的数组, M 控制曲线的平滑度, 其值越大, 曲线越平滑。



#### ClearScreen

语法: Procedure ClearScreen;

功能: 清屏。

#### ClearWindowstack

语法: Procedure ClearwindowStack(N: Integer);

功能: 清除第 N 个窗口。用这个过程清除的窗口不能用 Restorewindow 恢复。

#### Clipping

语法: Function Clipping; Boolean;

功能: 若允许剪辑, 返回 TRUE; 否则返回 FALSE。

#### CopyScreen

语法: Procedure CopyScreen

功能: 将活动屏幕上的图像传到非活动屏幕上。

#### CopyWindow

语法: Procedure CopyWindow(From, To: Byte; X, Y: Integer);

功能: 将活动屏幕的内容传到 RAM, 或将 RAM 中的内容传到活动屏幕。From 表示源屏幕, To 表示目标屏幕。为 1 时, 表示显示屏幕, 为 2 时表示 RAM, X, Y 表示窗口的屏幕位置。

#### DefineHeader

试法: Procedure DefineHeader(I: Integer; Hdr: Wkstring);

功能: 定义窗口 I 的标题。标题内容是 Hdr 字符串。

#### DefineTextWindow

语法: Procedure DefineTextWindow(I, Left, Up, Right, Down, Border: Integer);

功能: 用文本坐标 (80×25) 定义窗口 I。Left, Up, Right 和 Down 为窗口边界的文本坐标。Border 为窗口的文本区域外边界的象素数。

#### DefineWindow

语法: procedure Definwindow(I, Xlow, Ylow, Xhi, Yhi: Integer);

功能: 将屏幕的一个矩形区域定义为窗口 I。窗口位置由左上角坐标 Xlow, Ylow 和右下角的坐标 Xhi, Yhi 确定。

#### DefineWorld

语法: procedure DefineWorld(I: Integer; Xlow, Ylow, Xhi, Yhi, : Real);

功能: 定义 I 的完全坐标系。Xlow, Ylow 为屏幕的左上角, Xhi, Yhi 为右下角。坐标定义后, 需要用 Selectworld 命令实现。

DrawASCII/语法: procedure DrawASCII(Var X, Y: Integer; Scale, ch: Byte);

功能: 在绝对坐标 X, Y 处显示字符 ch, 字符扩大 scale 倍。

#### Drawaxis

语法: Procedure DrawAxis(XDensity, YDensity, Left, Top, Right, Bottom: Integer; XAxis, YAxis: Integer; Arrows: Boolean);

功能: 建立带刻度的横轴和纵轴。XDensity 和 YDensity 确定每个轴上出现的标记密度。Left, top, Right 和 Bottom 是绘图区域距屏幕边界的距离。XAxis 和 YAxis 确定轴的形式, 若为负数, 则不绘制其对应的轴。Arrows 若为 TRUE, 在轴的末端绘出箭头。

#### DrawBorder

语法: Procedure DrawBorder;

功能: 建立窗口的边界。

DrawcartPie

语法: procedure DrawCartPie(Xcenter, ycenter, Xstart, Ystart, Inner, Outer; Real; A: PieArray; N, Option, scale; Integer);

功能: 绘出以 Xcenter, Ycenter 为圆心, 从 Xstart, Ystart 开始的圆饼图, A 是 PieArray 形参数, 是由实数和标志组成的数组。本过程用 A 中的数计算圆饼图中每段相应的区域。N 是圆饼图的段数。Inner 和 Outer 分别表示圆饼图线的内端和外端, Option 确定标志或数字显示的方式: 0 不是显示标志; 1 显示文本标志; 2 显示文本和数字标志; 3 只显示数字标志。scale 是扩大或缩小标志的比例。

Drawcircle

语法: procedure Drawcircle(X, Y, R; Real);

功能: 在坐标 X, Y 处以半径 R 作圆。

Drawciyclesegment

语法: Procedure Drawcirclesegment(Xcenter, Ycenter; Real; Var Xstart, Ystart; Real; Inner, Outer, Angle, Area; Real; Text; Wrkstring; Option, Scale; Byte);

功能: 绘一个圆或圆弧。圆心位于 Xcenter, Ycenter。圆弧开始于坐标 Xstart, Ystart。这个过程也可以从园内到园外画一条线, 并在线的外端作一个标记。线的内端由 Inner 确定, 外端由 Outer 确定。area 表示标志部分显示的数字。Text 表示圆弧显示的标志。Option, Scale 与 Drawcircle 中的意义相同。

Drawline

语法: Procedure DrawLine(X1, Y1, X2, Y2; Real);

功能: 从坐标 AX1, Y1 到 X2, Y2 画一直线。

DrawLineClipped

语法: Procedure DrawLineClipped(X1, Y1, X2, Y2; Real);

功能: 从绝对坐标 X1, Y1 到 X2, Y2 画一直线。即使没有设置窗口模式, 在窗口边界处也可以截断此线。

DrowPoint

语法: Procedure Drawpoint(X, Y ; Real);

功能: 在坐标 X, Y 处画一像素。

DrawPolarPie

语法: Procedure DrawpolarPie(Xcenter, Ycenter, Radius, Angle, Inner, Outer; Real; A: PieArray; N, option, scale; Integer);

功能: 与 DrawCartpie 大致相同, 只是用 Radius 和 angle. Radius 是圆和半径的像素数, Angle 确定开始点的度数。

DrawPolygon

语法: Procedure DrawPolygon(A: PlotArray; First, Last, code, Scale, Lines; Integer);

功能: 通过数组 A 确定的一组点画线。First 和 Last 表示所画的起始顶点。Code 指示坐标处标号类型。Scale 指示标号显示的比例。Lines 控制填充线的显示。Lines 小于 0。从顶点到 X 轴

绘制填充线;大于0时,从底部到顶点画填充线。等于0时不画填充线。

#### DrawSquare

语法: procedure Drawsquare(X1,Y1,X2,Y2:real; Fill:Boolean);

功能: 画出一个由 X1,Y,和 X2,Y2 确定的长方形。长方形用当前线格式绘制。当 Fill 为 TRUE 时,用当前绘图颜色和模式填充。

#### DrawStraight

语法: Procedure Drawstraight(X1,X2,Y:Integer);

功能: 从 X1,Y1到 X2,Y 画一水平线。

#### Drawtext

语法: Procedure DrawText (X,Y,Scale:Integer;Text;Wrksting);

功能: 在绝对坐标 X,Y 处显示字符串 text,字符放大 Scale 倍。

#### DrawTextw

语法: Procedure DrawTexw(X,Y,Scale:Integer;Text;Wrkstring);

功能: 在完全坐标 X,Y 处显示字符串 Text,字符放大 Scale 倍。

#### EnterGraphic

语法: Procodure EnterGraphic;

功能: 清屏并设置图形方式。

#### Error

语法: Drocedure Error(Proc,Code);

功能: 在图形过程中发生错误时,error 显示出错信息。Proc 是发生错误的地址,Code 是错误类别,若允许中断,则停止程序执行。

#### Findworld

语法: Procedure FindWorld(I:Integer;APlotArray;N:INteger;Scalex,Scaley:real);

功能: 利用顶点数组 A 和顶点数 N, I 确定适当的完全坐标。Scalex 和 Scaley 是完全坐标扩大的比例。

#### GegAspect

语法: function GetAspect:Real;

功能: 返回当前纵横比的值。

#### Getcolor

语法: Function Getcolor:Integer;

功能: 返回当前绘图颜色的值。若返回值为0,则为背景色;若返回值为255,则为前景色。

#### getErrorcode

语法: Function GetErrorCode:Integer;

功能: 返回最近发生错误的代码。使用此函数时,必须用 SetBreakOff 禁止中断。下面是错误代码及其含义。

错误代码	含义
-1	无错
0	不可见错误信息
1	没有源文件

- 2 索引出界
- 3 坐标出界
- 4 数组元素太少
- 5 打开文件出错
- 6 超出窗口存储器
- 7 超域值

#### GetLineStyle

语法: Function GetLineStyle; Integer;

功能: 返回当前线格式。

#### Getscreen

语法: Function Getscreen; Integer;

功能: 返回值表示哪个屏幕是活动的: 1表示显示屏幕是活动的, 2表示 RAM 屏幕是活动的。

#### getScreenASpect

语法: Function GetscreenAspect; Real;

功能: 返回当前像素的纵横比。

#### GetVStep

语法: Function getVstep; Integer;

功能: 返回窗口中纵向移动一步的像素个数。

#### GetWindow

语法: Function GetWindow; Integer;

功能: 返回当前活动窗口的数目。

#### Hardcopy

语法: Procedure HardCopy(Inverse; Boolean; Mode; Byte);

功能: 打印屏幕上显示的图形。Inverse 为 TRUE 时, 反转打印(黑背景); 为 FALSE 时标准打印(黑前景)。Mode 决定打印图形的大小和密度。如下表

方 式	每行打印	epson 方式
0, 4, 5	640	4
1	960	1
2	960	2
3	1920	3
6	720	6

#### Hatch

语法: Procedure Hatch(X1, Y1, X2, Y2; Real; Delta; Integer);

功能: 用对角线填充屏幕上由坐标 X1, Y1 和 X2, Y2 确定的方形区域。斜线之间的距离由 Delta 确定。Delta 为正时, 划从左上到右下的斜线。为负时划从右上到左下的斜线。

#### InitGraphic

语法: Procedure InitGraphic;

功能: 初始化图形系统。只能调用这个过程一次。

#### INvertscreen

语法: Procedure InvertScreen;

功能: 将像素由黑变白, 或由白变黑。也可用于其它颜色。

#### Invertwindow

语法: Procedure InvertWindow;

功能: 与 Invertscreen 相似, 将窗口中的图像变反。

#### LoadScreen

语法: Procedure LoadScreen(FileName; WrkString);

功能: 从文件 FileName 中将一个屏幕装入存储器。

#### Loadwindow

语法: Procedure LoadWindow(N, X, Y; Integer; FileName; Wrkstring);

功能: 从文件 FileName 中读出一个窗口, 将其放入窗口 N, 在坐标 X, Y 处显示。

#### LoadWindowStack

语法: Procedure LoadwindowStack(FileName; WrkString);

功能: 读入两个文件: FileName.STK 和 FileName.PTR。它们是以以前存入的窗口堆栈。

#### LeaveGraphic

语法: Procedure LeaveGraphic;

功能: 返回到文本方式。

#### MoveHor

语法: Procedure MoveHor(Delta; Integer; Fillout; Boolean);

功能: 将活动窗口水平移动 Delta。当 Delta 为负时左移, 为正时右移。Fillout 决定窗口移动后的空间的变化。当 Fillout 为 FALSE 时, 为当前颜色的反色。为 TRUE 时, 为当前颜色。

#### MoveVer

语法: Procedure MoverVer(Delta; Integer; Fillout; Boolean);

功能: 垂直移动活动窗口。Delta 为正时, 向上移动, 为负下移。Fillout 的意义同 MoveHor。

#### PointDrawn

语法: Function PointDrawn(X, Y; Real); Boolean;

功能: 若点是用完全坐标 X, Y 绘制的, 则返回 TRUE。

#### RedefineWindow

语法: Procedure RedefineWindow(I, Xlow, Ylow, Xhi, Yhi; Integer);

功能: 改变窗口 I 的大小。坐标 Xlow, Ylow 确定左上角, Xhi, Yhi 确定右下角。

#### RemoveHeader

语法: Procedure RemovdHeader(I; Integer);

功能: 去掉窗口 I 的标题。

#### ResetWindows

语法: Procedure ResetWindows;

功能：将所有窗口设置为屏幕大小，并将窗口 1 作为活动窗口。

ResetWindowStack

语法：Procedure ResetWindowstack;

功能：删除窗口堆栈中的窗口并释放内存。

ResetWorlds

语法：proceddure ResetWorlds;

功能：将所有完全坐标初始化为屏幕绝对坐标。

RestoreWindow

语法：Procedure RestoreWindow(N,X,Y;Integer);

功能：从窗口堆栈中取出窗口 N，在坐标 X，Y 处显示它。

RotatePolygon

语法：Procedure RotatePolygon(A;PlotArray; N;Integer; Angle;Real);

功能：将由 A 和 N 决定的多边形旋转 Angle 度，旋转中心由图形工具箱计算。

RotatePolygonAbout

语法：Procedure RotacPolygonAbout(A;Plot.Array; N;INteger; Angle,X,Y;Real);

功能：将由 A 和 N 决定的多边形旋转 Angle 度，旋转中心由完全坐标 X,Y 确定。

SaveScreen

语法：Procedure SaveScreen(FileName;Wrkstring);

功能：将图形存入 FileName 指定的文件中。

saveWindow

语法：Procedure SaveWindow(N;Integer;File Name;wrkstring);

功能：将窗口 N 存入 FileName 文件中。

SavewindowStack

语法：procedure SaveWindowStack(FileNmac;Wrkstring);

功能：将所有窗口存入 FileName.STK 和 FileName.PTR 两个文件中。扩展名由图形工具箱添加。

Scalepolygon

语法：Procodure Scalepolygon(A;PlotArray;N;Integer; Xfactor,Yfactor;Real);

功能：将 A 的 N 个坐标 X，Y 分别乘以 Xfactor 和 Yfactor。这将改变多边形的显示比例。

SelectScreen

语法：Procedure SelectScreen(I;Integer);

功能：图形工具箱允许在活动屏幕(可见屏幕)或 RAM 屏幕(在存储器中，不可见)上显示图形。当 I 为1 时，选择活动屏幕，为2时选择 RAM 屏幕。

Selectwindow

语法：Procedure Selectwindow(N;Integer);

功能：将窗口 N 变为活动窗口。

SelactWorld

语法：Proccodure SelectWorld(I;Integer);

功能：选择完全坐标系 I。

SetAspect

语法: Procedure SetAspect(Aspect;Real);

功能: 改变显示园和椭园的纵横比, 用 aspect 乘改变纵比。如 SetAspect(2)。显示的园, 高二倍于宽。

SetBackground

语法: Procedure SetBackground(Pattern ;Byte);

功能: 用 Pattern 设置背景模式。

SetBackground8

语法: Procedure SeBackground8(Pattern;BackgroundArray);

功能: 由 Pattern 确定一个背景图形。Pattern 是8字节的数组, 定义屏幕上8×8个象素的一个区域。

SetBackgroundcolor

语法: Procedure setBackgroundcolor(Color;Integer);

功能: 设置背景象素的颜色。

SetBreakoff

语法: Procedure SetBreakOff;

功能: 禁止中断方式, 若出现错误条件将使程序失败。

SetBreakOn

语法: Drocedure SetBreakOn;

功能: 允许中断方式。若出现错误条件, 中断方式会结束程序。

SetcolorBalck

语法: Procedure SetColorBLack;

功能: 选择当前背景色为绘图颜色。

SetColorWhite

语法: Procedure SetColorWhite;

功能: 选择当前前景色为绘图颜色。

SetclippngOff

语法: Procedures Setclipping—Off

功能: 禁止剪辑方式。这时允许在当前活动窗口边界之外的区域绘图。

SetClippingOn

语法: Procedure SetClippingOn;

功能: 允许剪辑方式。这时只能在活动窗口边界之内绘图。

SetForegroundcolor

语法: procedure SetForegroundColor(color;Integer);

功能: 设置前景象素的颜色。

SetHeaderOff

语法: Procedure SetHeaderOff;

功能: 在使用 DrawBorder 命令时不显示窗口标题。

SetHeaderOn

语法: Procedure SetHeaderOn;

功能: 在使用 DrawBorder 命令时, 显示窗口标题。

#### SetHeaderToBottom

语法: Procedure StHeadertoBotom;

功能: 在窗口底部显示标题。

#### SetHeaderToTop

语法: Procedure SetHeadertoTop;

功能: 在窗口顶部显示标题。

#### SetLineStyle

语法: Procedure SetLineStyle(Ls: Integer);

功能: 确定绘线类型。Ls 可以指定 5 种线类型。0—实线; 1—点线; 2—破折号; 3—破折号和点线; 4—短破折号。

#### SetMessageOff

语法: procedure SetMessageOff;

功能: 减少工具箱显示的错误信息。当设置中断方式时, 缩短出错信息。当禁止中断时, 不显示出错信息。

#### SetMessageOn

语法: procedure SetMessageOn;

功能: 产生扩展的错误信息。当设置中断方式时, 显示完整的错误信息并停止程序。当禁止中断时, 在第 24 行上显示错误信息, 但不停止程序的执行。

#### SetScreenAspect

语法: Procedure SetScreenAspect(Aspect: Real);

功能: 决定像素的横比。

#### SetVstep

语法: Procedure SetVstep(Step: Integer);

功能: 设置窗口每步移动的纵向距离。Step 是个正整数, 确定这个距离的像素数。

#### SetWindowModeOff;

语法: procedure SetWindowModeOff;

功能: 只允许使用绝对坐标绘图。

#### SetWindowModeOn

语法: procdeure SetWindowModeOn;

功能: 设置窗口模式, 这允许用完全坐标绘图。

#### Spline

语法: Procedure Spline(A: PlotArray; N: Integer; Xl, Xm: Real; Var B: PlotArray; M: Integer);

功能: 在由 A 和 N 确定的多边形顶点之间插入一条平滑曲线, 并将结果存入 B 中。Xl, Xm 表示 A 中的起点, M 是定义平滑曲线的点数。

#### StoreWindow

语法: Procedure StoreWingow(N: Integer);

功能: 将窗口 N 存入窗口堆栈。

#### Swapscreen

语法: Procedure Swapscreen;



功能：交换显示屏幕和 RAM 屏幕。

Translatepolygon

语法：Procedure Translatepolygon(A:PlotArray; N:Integer;DeltaX, Deltay:Real);

功能：通过将 A 和 N 确定的多边形顶点 X 坐标加 DeltaX, Y 坐标加 Deltay 来移动多边形。

windoMode

语法：Function WindowMode:Boolean;

功能：若当前设置为窗口模式，返回 TRUE。

WindowSize;

语法：Function windowSize (N:Integer);

功能：返回窗口堆栈中存贮窗口 N 所需的内存。内存大小以千字节为单位。

WindowX

语法：Function WindowX(X:Real);Integer;

功能：返回完全坐标 X 在屏幕绝对坐标上对应的值。

Windowy

语法：Function Windowy(Y:Real);Integer;

功能：返回完全坐标 在屏幕绝对坐标上对应的值。

### § 8.3 编辑工具箱

计算机的一项主要功能是进行字处理。对字处理程序的一个主要要求是速度。Turbo Pascal 编辑工具箱的速度非常快。它使用了链表缓冲区结构。Turbo Pascal 编辑工具箱的另一个特点是提供了大量处理功能。

编辑工具箱有三类字处理器：二进制数编辑，First—Ed 和 MicroStar。下面主要介绍 First—Ed 和 MicroeStar。下面是编辑工具箱中的部分过程和函数：

EdAbandonFile

语法：Procedure EdAbandonFile(ExitEditor:Boolean);

功能：不存盘退出当前编辑的文件。

EdArg2Integer

语法：Procedure EdArg2Integer(Arg:String255, Min, Max:Integer;Var V);

功能：将串 Arg 变成介于 Min 和 Max 之间的整数，放入 V 中。

EdBackTab

语法：Procdure EdBackTab;

功能：实现一个向后制表命令。

EdBackupCurLine

语法：Procedure EdBackupCurLine(W:PwinDesc);

功能：当缩小一个窗口时，使光标上移，确保光标在窗口中。

EdBiosScroll

语法：Procedure EdBiosScroll;

功能：用 BIOS 功能使屏幕区域上下滚动。

#### EdBlockBegin

语法: Procedure EdBlockBegin;

功能: 在光标处设置块起始标志。

#### EdBlockCopy

语法: Procedure EdBlockCopy;

功能: 块拷贝命令处理。

#### EdBlockDelete

语法: Procedure EdBlockDelete;

功能: 块删除命令处理。

#### EdBlockEnd;

语法: Procedure EdBlockEnd;

功能: 在光标处设置块结束标志。

#### EdBlockInit

语法: Procedure EdBlockInit;

功能: 在块内进行搜索时, 将光标置于块的开始处(向后搜索)或结尾处(向前搜索)。

#### EdBlockMove

语法: procedure EdBlockMove;

功能: 将块移到光标处。

#### EdBlockRead

语法: Procedure EdBlockRead(Var F;File; Var Buf; Num; Word; Var BytesRead);

功能: 从文件 F 中读出 Num 字节的正文送入缓冲区 Buf 中, BytesRead 是实际读取的字节数。

#### EdBlockWord

语法: Procedure EdBlockWord;

功能: 对光标右边的字作块标记。

#### EdBlockWrite

语法: Procedure EdBlockwrite(Var F;File;Var Buf; Num;word);

功能: 将缓冲区 Buf 中的 Num 个字节的正文写入文件 F 中。

#### EdBottomScreen

语法: procedure edBottomScreen;

功能: 将光标移到当前屏幕的最后一行。

#### EdBreathe

语法: Proecedure EdBreathe;

功能 从 BIOS 缓冲区取字符, 放入编辑器的大容量缓冲区。

#### EdBuffercurrentLine

语法: Procedure EdBufferCurrentLine;

功能: 把当前行存贮起来, 以后可用 QAL 命令恢复此行。

#### EdBufferSize

语法: Function EdBuferSize(Ncols;Integer);Inteter;

功能: 返回 Ncols 正文所需的内存量。

#### EdCalcMemory

语法: Function EdCalcMemory: VarString;

功能: 用串的形式返回堆中可用存贮器的数量。

#### EdCenterLine

语法: Procedure EdCenterLine;

功能: 将当前行置于中间位置。

#### EdChangeCase

语法: procedure EdchangeCase(Mode; CaseChange);

功能: 改变字符的大小写。

#### EdChangeStreamName

语法: Prcedre EdchangeStreamName(Fname; Filepath);

功能: 将当前窗口正文流名改为 Fname。

EdChooseappending/语法: Procodure EdchooseAppending(Var Choise, Integer);

功能: 显示菜单, 询问是重写文件还是在文件后添加正文。

#### EdClassifyInput

语法: Procedure EdclassifyInput;

功能: 从字符缓冲区中取得下一个字符, 放入正文流中。若为控制字符, 则调用适当的程序。

#### EdCleanFileName

语法: procedure EdCleanFileName(Var Fname; Filepath);

功能: 接受文件名 Fname, 将其变为大写字母, 删除空格, 扩展文件路径名。并进行某些错误检查。

#### EdClearBuffer

语法: Procoduer EdclearBuff

功能: 清空键盘缓冲区。

#### Edclearstring

语法: procedure EdClearString(Var s);

功能: 将串 S 置为空串。

#### EdCloneModifiedFlags

语法: Procedure EdCloneModifiedFlags

功能: 编辑器允许在多个窗口观察同一正文流。这个过程保证在一个窗口修改的标志在其它窗口中也得到修改。

#### EdCompress

语法: procedure Edcompress(P; PlineDesc; Lmargin ; Integer; Var Col, Len; Integer);

功能: 在做行的右调整时, 要加入一些空格。这个过程将删除这些空格。

#### EdComPuteEffectiveCoLNo

语法: Function EdComputeEffectiveCoLNo(Attribs; Boolean;

D; PlineDesc; Col; INteger); Integer;

功能: 返回 P 行的列数 Col。当 Attribs 为 TRUE 时, 返回有效列数, 否则返回 Col 的初值。

EdControlFilter

语法: Function EdControlFilter(Ch;Char):Char;

功能: 返回与字符 Ch 对应的控制字符。如 EdControlFilter('L') 返回 CTRL—L。

EdCursorInBlock

语法: Function EdCursorInBlock(Q:PlineDesc;C:Interreg;End—MarkOn:Boolean):  
Boolean;

功能: 从块头开始搜索正文缓冲区, 确定 Q 行 C 列是否在在块中。如果 EndMarkon 为 FALSE, 并且 Q 行 C 列恰好是模块结束位, 则也返回 FALSE。

EdDefaultextension

语法: procedure EdDefaultextension(Ext;Var String;Var Fname;Filepath);

功能: 若 Fname 不带扩展名, 则将 Ext 加到其后。

EdDeleteAllText

语法: Procedure EdDeleteAllTex(W:PwinDesc);

功能: 删除整个正文流。

EdDeleteLeadingBlanks

语法: Procedure EdDeleteLeadingBlanks(Var S:String);

功能: 删除字符串 S 前面的空格。

EdDeleteLeftChar

语法: procedure EdDeleteLeftchar;

功能: 删除光标左边的字符。

EdDeleteLine

语法: Procedure EdDeleteLine;

功能: 删除当前行, 不放入可恢复缓冲区中。

EdDeleteLineRight

语法: Procedure edDeleteLineRight;

功能: 删除当前行光标右边的所有正文。

EdDeleterightChar

语法: Procedure EdDeleteRightChar;

功能: 删除光标右边的一个字符。

EdDeleteRightWord

语法: Procedure EdDeleteRightWord;

功有: 删除光标右边的一个字。

EdDeleteTrailers

语法: procedure EdDeleteTrailers(Var S:String);

功能: 删除串 S 中第一个字后的所有字符。

EdDirectory

语法: Procedure EdDirectory;

功能: 显示目录文件。

语法: Procedure EdDownpage;

功能: 光标下移一页。

### EdDrawBox

语法: `procedure EdDrawBox (Border: BorderChar, Xposn, Yposn, Xsize, :Byte; BoxAttr: BoxType);`

功能: 以 Xposn, Yposn 为左上角画一方框, 水平为 Xsize 个字符, 垂直为 Ysize 个字符。边界为 Boder, 类型为 BoxAttr。

### EdDrawItem

语法: `Procedure EdDrawItem (Menu: Menuptr; SubBypte);`

功能: 在菜单中显示选择项。

### EdDrawMenu

语法: `Procedure EdDrawMenu (Menu: Menuptr);`

功能: 显示菜单。

### EdEditTabLine

语法: `Procedure EdEditTabLine`

功能: 编辑制表行。

### EdEraseMenuHelp

语法: `Procedure EdEraseMeauHelp;`

功能: 删除帮助信息。

### EdEraseMenus

语法: `Procedure EdEraseMenus; /功能: 删除所有菜单。`

### EdErrorMsg

语法: `Procedure EdERrorMsg (Msgno: Integer);`

功能: 显示错误信息, 并清除字符缓冲区。

### EdFileWrite

语法: `Procedure EdFileWrite (Fname, Filepath; Quitling: Boolean);`

功级: 将当前正文流存入文件 Fname, 若 Quitling 为 TRUE, 则程序结束。

### EdFixBlockInsertedSpace

语法: `procedure EdFixBlockInsertedSpace (P: PlineDesc; Start: Integer; num: integer);`

功能: 在 P 行中从 Start 处开始删除 Num 个字符。若 Num 为正, 则增加, 为负则删除。

### EdGetCustomMenuChoice

语法: `Procedure EdGetCustomMenuChoice (Var Menu; CustomMenuREc) vAr. Choce; Integer);`

功能: 显示一个菜单并返回选择。

### EdGetMessage

语法: `Function EdGetMessage (MsgnoL: Integer); String;`

功能: 从压缩信息缓冲区中返回信息。

### EdGetOptions

语法: `Procedure EdGetOption (Xp, Yp, width, MaxLen; Integer; Have Window; Boolloan);`

功能: 要求键入搜索选择项, 进行查找或替换。

### EdGetSearchString

语法: `procedure EdGetSearchString (Xp, Yp, Width, MarLen; Integer; HaveWindow;`

Boolean; Var SearchStr; VarString);

功能：要求输入要搜索的串。

EdGlobalInit

语法：Procedure EdGlobalInit;

功能：将光标置于正文流的开始。

EdGotoColumn

语法：procedure EdGotoColumn(Cno;Integer);

功能：将光标放到 Cno 处。

EDGotoPage

语法：Procedure EdGotoPage(Pno;Integer);

功能：将光标移到 Pno 页。

EdGotoXY

语法：Procedure EdGotoXY(C,R;Byte);功能：将光标移到 R 行 C 列。

EdHscroll

语法：procedure EdHscroll;

功能：窗口垂直滚动。

EdInsertLine

语法：Procedure EdInsertline;

功能：插入一行。

EdInsertSpace

语法：Function EdInsertSpace(P;PlineDesc;Start;Integer; Num;Integer):Boolean;

功能：在行 P 行 Start 处插入 Num 个空格。

EdInsertundoBuffer

语法：procedure EdInsertundoBuffer;

功能：在光标处插入 undo 缓冲区的内容。

EdJumpMarker

语法：procedure EdJumpMarker(M;BlockMarRer);

功能：将光标移到标志 M 处。

EdleftLine

语法：Procodure EdLeftLine;

功能：将光标置于行的最左边。

EdLeftword

语法：Procedure EdLeftWord;

功能：将光标移到左边字的开始。

EdLineindent

语法：FurCtion EdLineIndent(P;PlineDesc):Integer;

功能：返回 P 行中的空格数，若空行则返回0。

EdLinkBuffer

语法：Procedure EdLinkBuffer(P,Q;PLineDesc);

功能：将 Q 行接在 P 行后。

#### EdLocase

语法: Procedure EdLocase(Var S:String);

功能: 将 S 中的大写字母变为小写。

#### EdLongPosBack

语法: Function EdLongPosBack(Var Buffer; Start :Integer; Var Pattern;Var String):Integer;

功能: 从 Start 处开始向后搜索 Pattern.若找到, 返回其位置.否则返回0。

#### EdLongPosFwd

语法: Function EdLongPosFwd (Var Buffer; Start, Size; Integer; Var Pattern; VarString):Integer;

功能: 与 EdLongPOsBack 相同, 只是向前搜索。

#### EDLongUpcase

语法: Procedure EdLongUpcase(Var Buffer;Size;Integer);

功能: 将 Buff 中的 Size 个字符变为大写字母

#### EdMakeBekFile

语法: Procedure EDMakeBakFile(Fname;Filepath);

功能: 建立 Fname 的备份文件。

#### EdMoveToBegin

语法: Procedure EdMoveToBegin;

功能: 光标移到前一行。

#### EdMoveToEnd

语法: Procedure EdMoveToEnd;

功能: 光标移到下一行。

#### EdNewLine

语法: Procedure EdNewLine;

功能: 回车换行。

#### EdNextSentence

语法: Procedure EdNextSentence;

功能: 光标移到下一句子开头。

#### EdPrevSentence

语法: Procedure EdPrevSetence;

功能: 光标移到上一句子的开头。

#### EdProcessCommand

语法: Procedure EdProcessCommand(C;CommandType);

功能: 编辑器的主命令处理程序。

#### EdPromptWriteBlock

语法: Procedure EdPromptWriteBlock;

功能: 输入文件名, 将正文写入这个文件。

#### EdPushUndo

语法: Procedure EdPushUndo(Var P;PlineDesc);

功能：将 P 行送入 Undo 中。

EdReadFile

语法：Procedure EdReadFile(Fname; Filepath);

功能：检查并打开文件 Fname 进行编辑。

EdReadtextFile

语法：Procedure EdReadTextFile(Fname; FilePath; ReadingBlock; Boolean);

功能：将文本文件 Fname 读入当前窗口。如果 ReadingBlock 为 PRUE，则读入的文件为带标志的块，否则为新的正文流。

EdRealign

语法：Procedure EdRealign;

功能：重新定向窗口的正文流。

EdRealignOne

语法：procedure EdrealignOne(W; PwinDesc);

功能：重新定向窗口。

EdReformBlock

语法：Procedure EdReformBlock;

功能：重新格式块。

EdReformParagraph

语法：Procedure EdReformParagraph;

功能：重新格式当前段。

EdResetWindow

语法：Procedure EdReSetWindow(W; PwcnDesc);

功能：打开一个新窗口，并复位窗口描述符。

EdRestoreScreenMode

语法：Procedure EdRestoreScreenMode;

功能：退出时清屏。

EdRightLine

语法：Procedure EdRightLine;

功能：光标移到行右界。

EdRightWord

语法：Procedure EdrightWord;

功能：光标移至下一个字。

EdSaveFile

语法：Procedure edsaveFile;

功级：正文存盘并继续编辑。

EdScrollDown

语法：Procedure EdscrollDown;

功能：下滚。

EdScrollup

语法：Procedure EdScrollUp;



功能：上滚。

EdSetColor

语法：procedure EdSetColor;

功能：设置编辑器颜色。

EdSetega25LineMode

语法：procedure EdSetega25LineMode;

功能：设置 EGA 卡为 25 行显示。

EdSetega43LineMode

语法：Procedure EdsetEga43LineMode;

功能：设置 EGA 卡为 43 行显示。

EdSetLeftMargin

语法：procedrue EdSetLeftMargin;

功能：设置左空白。

EdSetupWindow

语法：Function EdSetupwindow (Border; BorderChars; XLow, YLow, Xhigh, Yhight;  
Byte; BoxAttr; BpxType); WindowPtr;

功能：存贮当前屏幕并建立一个新窗口

EdShowMemory

语法：Procedure EdshowMemory;

功能：显示文本可用的自由内存。

EdSizeLine

语法：Function EdSizeLine(P; PlineDesc; Ncols; Integer; Init Boolean); Boolean;

功能：扩大行使其可纳 Ncols 个字符。

EdSpellingCheck

语法：Procedure EdSpelling—Check;

功能：检查文件或模块的拼写。

EdString2Integer

语法：Procedure EdString2Integer(Src; Var String; Var R);

功能：将串化为整数。

EdSysInfo

语法：Procedure EdSysInfo;

功能：显示编辑器信息。

EdTextlength

语法：Function EdtextLength(P; PLineDesc); Integer;

功能：返回正文行长度。

EdToPofStream

语法：Function EdtopofSream(W; PwinDesc); PLineDesc;

功能：返回正文流中第一行的指针。

EdTopscreen

语法：Procedure EdTopScreen;

功能：将光标移至屏幕顶部。

EdUppcase

语法：procedure EdUppcase(Var S:String);

功能：将串的小写字母变为大写。

EDUpdateCursor

语法：procduure EdUpdateCursor;

功能：光标右移一格。

EdUpline

语法：prcedure EdUpline;

功能：上移一行。

EdUppage

语法：procedure EdUpPage;

功能：上移一页。

EdWindowBottomFile

试法：prcedrue EdwindowBottomFile;

功能：将光标移至文件末尾。

EdWindowDelete

语法：procedure EdWindowDelete(Wno:Byte);

功能：删除窗口。

EdWindowDown

语法：Procedure EdwindowDown;

功能：窗口下移。

EdWindowUp

语法：proceduer EdWindowUp;

功能：窗口上移。

EdZoomwindow

语法：Procedure Zoomwindow(FileCurLine:Boolan);

功能：使当前窗口占据整个屏幕。

ExecShrink

语法：Function ExecShrink(Command:String):Integer;

功能：运行 DOS 命令。

## § 8.4 数值方法工具箱

Turbo Pascal 的数值方法工具箱提供了一套常用数值算法。

用二分法求函数的根：

语法：procedure Biset(LeftEnd:Real;Right:Real; Tolerance:Real; MaxIter:Integer; Var  
                          Answer:Real; Var fAnswer:Real; Var Iter:Integer; Var Error;  
                          Bter);

说明：计算 TNTargetF 中实连续函数的根

输入参数:LeftEnd 区间的左端点, RightEnd 区间的右端点, Tolerance 允许误差, MaxIter 最大迭代次数, 输出参数:Answer, 近似根, fAnswer 为在近似根处函数的值, Iter 为迭代次数。

牛顿—拉福森方法求方程的根:

语法: procedure Newton—Raphson(Guess:Real; Tolerance:  
Real; MaxIter:Integer;  
Var Root:Real;  
Var Value:Real;  
Var Deriv:Real;  
Var Iter :Integer;  
Var Error:Byte);

说明: 用牛顿—拉福森方法求 TNTargetF 函数的根。输入参数:Guess 为根的近似; Tolerance 允许误差, MaxIter 最大迭代次数。输出结果:Root 近似根, Value 近似根处的函数值, Deriv 近似根处的导数, Iter 迭代次数。

割线法求函数的根。

语法: procedure Secant(Guess1:Real;Guess2:Real;  
Tolerance:Real;MaxIter:Integer;  
Var Root:Real; Var Value:Real;  
Var Iter,Integer;Var Error:Byte);

说明: 给定两个初值 Guess1, Guess2, 用割线法计算 TNTargetF 函数的根。最大迭代次数 MaxIter, 允许误差为 Tolerance, Root 为近似根, Value 为函数在近似根处的值, Iter 迭代次数, error 为错误代码。

带压缩牛顿—霍纳法求实多项式方程根

语法: procedure Newt—Horn—Def1(InitDegree:Integer;  
InitPoly:TNvector;  
Guess:Real;  
ToBerance:Real;  
MaxIter:Integer;  
Var Degree:Integer;  
Var NumRoots:Integer;  
Var Poly:TNvector;  
Var Root:TNvector;  
Var Imag:TNvector;  
Var Value:TNvector;  
Var DeriV:TNvector;  
Var Iter:TNvector;  
Var Error:Byte);

说明: 用初始值 Guess 估计用户说明多项式的几个根, InitDegree 为多项式的次数, InitPoly 多项式的系数, Tolerance 允许误差, MaxIter 最大迭代次数, Degree 为压缩多项式的次数, NumRoots 近似根的个数, Poly 近似根的实部, 近似根处多项式的值, deriv 近似根处的导数。

Iter 迭代次数, error 错误代码。

Muller 法求复函数的复数根

语法: procedure Mullor (Guess: TNcomplex; Tolerance: Real;  
MaxIter: Integer; Var Answer: TNcomplex; Var yAnswer:  
TNcomplex;  
Var Iter: Integer; Var Error: Byte),

说明: 用 Muller 方法求函数 TNTargetF 的近似复数根。Guess 为初始估值, Tolerance 为允许误差, MaxIter 最大迭代次数, Answer 为求出的近似根, yAnswer 为在近似根处的函数值, Iter 为迭代次数, Error 为错误代码。其中 TNcomplex 为

TNComplex = Record

Re,

Im: Real;

End;

TNTargetF 为

procedure TNTarget (X: TNcomplex;

var Y: TNcomplex);

Laguerre 压缩法求复多项式的根。

语法: procedure Laguerre (Var Degree: Integer; Var Poly:  
TNcompvector; InitGuess:  
TNcomplex; Tolerance: Real;  
MaxIter: Integer; Var NumRoots:  
Integer; Var Roots: TNcompvector;  
Var YRoots: TNcompvector;  
Var Iter: TNIntvector;  
Var error: Byte)

说明: 用 Laguerre 法求复多项式的根。根可以是复数, Degree 为多项式的次数, Poly 为多项式的系数, InitGuess 为初值, Tolerance 为允许误差, MaxIter 为最大迭代次数, NumRoots 近似根的个数, roots 为近似根, YRoots 为近似根处多项式的值, Iter 迭代次数, Error 为错误代码。

Lagrange 插值法

语法: procedure Lagrange (NumPoints: Integer;  
Var Xdata: TNvector;  
Var Ydata: TNvector;  
NumInter: Integer;  
Var XInter: TNvector;  
Var YInter: TNvector;  
Var Poly: TNvector;  
Var Error: Byte);

说明: 用 Lagrange 方法求一个多项式拟合数据 X 和 Y, 并对 XInter 的 X 数据点插入 Y 数据点, NumPoints 为数据点数, XData 为 X 数据点, YData 为 Y 数据点, NumInter 为插入

次数。XInter 为插入的 X 数据点, YInter 为插入的 Y 数据点。Poly 为插值多项式的系数。error 为错误代码。

#### 自由立方样条插值

语法: Procedure Cubicspline--Free(NumPint; Integer; Var Xdata; TNvector;  
Var YData; TNvector;  
NumInter; Integer;  
Var XInter ; INvector;  
Var Coef0 : TNvector;  
Var Coef1 : TNvector; Var coef2 : TNvector;  
Var Coef3 : TNvector;  
Var YInter : TNvector;  
Var error : Byte);

说明: 求通过一数据点集的平滑曲线。NumPoints 为数据点个数, XData 为 X 数据点, YData 为 Y 数据点, NumInter 为插值次数, XInter 为插值处 X 数据点。Coef0 为常数项, Coef1 一次项系数, Coef2 为二次项系数, Coef3 为三次项系数, YInter 插值的 Y 数据点。Error 错误代码。

#### 强制三次样条插值

语法: Procedure CubicsplineClamped(Numpoints; Integer;  
Var XData ; TNvector;  
Var YData ; TNvector;  
DeriveLE : Real;  
DeriveRE : Real;  
NumInter ; Integer;  
Var XInter ; TNvector;  
Var Coef0 : TNvector;  
Var Coef1 : TNvector;  
Var Coef2 : TNvector;  
Var YInter : TNvector;  
Var Error : Byte);

说明: 对给定的数据点 X 求出一系列数据点 Y, 使得到的数据点 X 和 Y 形成过原数据点 X 和 Y 的光滑曲线。其本身及一阶和二阶导数也是连续的。Numpoints 为数据点个数, XData 为 X 数据点, YData 为 Y 数据点, DeriveLE 为左端点的导数, DeriveRe 为右端点的导数, NumInter 插值数据点的个数。XInter 要插入处的 X 数据点。Coef0 为常数项, Coef1 为一次项系数, Coef2 为二次项系数, Coef3 为三次项系数, XInter 为插入值。Error 为错误代码。

#### 一阶微分

语法: Procodre First--Derivative(Numpoints; Integer;  
Var XData ; TNvector;  
Var YDta : TNvector;  
Point : Byte;  
NumDeriv : Integer;

```

Var XDeriv ; TNvector;
Var YDeriv ; TNvector;
Var Error ; Byte);

```

说明：用一组 X 数据和 Y 数据求函数  $Y=f(X)$  的一阶导数的近似值。NumPoints 数据个数 XData 为 X 数据组，YData 为 Y 数据组。Point 为点的个数，必须为 2, 3 和 5。NumDeriv 近似导数处点的个数，XDeriv 为近似导数处 X 数据的个数，YDeriv 为每个 X 的近似 Y 数据，Error 为错误代码。

### 二阶微分

语法：Procedure Second-Derivative(NumPoints; Integer;

```

Var XData; TNvector;
Var YData ; TNvector;
Point ; Byte;
NumDeriv ; Integer;
Var Xderiv; TNvector;
Var YDeriv; TNvector;
Var Error ; Byte);

```

说明：求函数  $Y=f(x)$  的二阶导数的近似值。Numpoint 数据组个数。XData 为 X 数据组，YData 为 Y 数据组，Point 点数，必须为 3 或 5。NumDeriv 近似导数点的个数，XDeriv 近似导数的 X 数据。YDeriv 为用 XDeriv 中 X 求出的 Y 数据，Error 错误代码。

### 三次样条插值微分

语法：Procedure Interpolate-Derivative(Numpoints; Integer;

```

Var Xdata ; TNvector;
Var YData ; TNvector;
NumDeriv ; Integer;
Var Xderiv ; TNvector;
Var Yinter ; TNvector;
Var YDeriv ; TNvector;
Var YDeriv2; TNvector;
Var error ; Byte);

```

说明：求函数  $Y=f(x)$  的近似一阶和二阶导数。Numpoints 数据个数，XData 为 X 数据组，YData 为 Y 数据组，NumDeriv 求近似导数处 X 点的个数。

XDeriv 为 X 的数据。YInter 为 X 点处 Y 的近似值。Deriv 为一阶导数，Deriv2 为二阶导数。Error 为错误代码。

### 用户定义函数的一阶微分

试法：Procedure FirstDerivative(NumDeriv ; Integer;

```

Var XDeriv; TNvector;
Tolerance; Real;
Var Error ; Byte);

```

说明：用 X 点集合求函数  $y=f(x)$  的近似一阶导数。用户定义的函数为：

```
Function TNtarget(x; Real); Real;
```

NumDeriv 为求导数处的点个数。XDeriv 为 X 点, tolerance 为允许误差, YDeriv 为 X 点处的近似一阶导数, Error 为错误代码。

用户定义函数的二阶微分

语法: Procedure SecondDerivative(NumDeriv: Integer;  
Var Xderiv: TNvector;  
Var YDeriv: TNvector;  
Tolerance: Real;  
Var Error: Byte);

说明: 用 X 点求  $y=f(x)$  的近似二阶导数。用户定义函数同前, NumDeriv 用于求近似导数的点个数, XDeriv X 点集, Tolerance 解的精度, YDeriv 为 X 点的近似二阶导数, Error 错误代码。

辛普森算法求积分

语法: procedure Simpson(LowerLimit: Real; UpperLimit: Real; NumIntervals: Integer;  
Var Integral: Real;  
Var Error: Byte);

说明: 计算在 LowerLimit 和 UpperLimit 这之间的用户定义函数的积分, NumIntervals 求近似积分的区间数, Integral 函数的近似积分, Error 错误代码。

用梯形复合规则求积分

语法: Procedure Trapeziod(LowerLimit: Real; UpperLimit:  
Real; NumIntervals: Integer;  
Var Integral: Real;  
Var Error: Byte);

说明: 计算在 LowerLimit 和 UpperLimit 这间, 用户定义函数的积分, NumIntervals 为区间的个数。

Integral 为积分值, Error 为错误代码。

用自适应面积法和辛普森法则求积分

语法: Procedure Adaptive—Simpson(LowerLimit: Real;  
UpperLimit: Real;  
Tolerance: Real;  
MaxIntervals: Integer;  
Var NumIntervals: Integer;  
Var Error: Byte);

说明: 在 LowerLimit 和 UpperLimit 之间求积分, Tolerance 为解的精度, MaxIntervals 为允许的最大区间数, Integral 为积分近似值, Error 为错误代码, NumIntervals 为实际区间数。

自适应高斯面积法求积分

语法: procedure Adaptive—Gauss—Quadralure(LowerLimit:  
Real;  
UpperLimit: Real;  
Tolerance: Real;  
MaxIntervals: Integer;

```

Var Integral ; Real;
Var NumIntervals; Integer;
Var Error; Byte);

```

说明：在 LowerLimit 和 UpperLimit 之间求函数的积分值。Tolerance 为解的精度，Max-InterVals 为最大区间数。Integral 为求出的积分值。NumInterVals 为实际区间数，Error 为错误代码。

Romberg 法求积分

```

语法：Procedure Romberg(LowerLimit; Real; MexIter;
                          Real; Tolerance; Real; MaxIter;
                          Integer; Var Integral ; Real;
                          Var Iter; Integer; Var Error;
                          Byte);

```

说明：求函数在 LowerLimit 和 UpperLimit 之间的积分值。tolerance 为解的精度，MaxIter 为最大迭代次数。Integral 为求出的积分值，Iter 为迭代次数。Error 为错误代码。

求矩阵的行列式

```

个法：Procedure Determinant(Dimen; Integer; Data;
                              TNmatrix; Var Det; Real;
                              Var error; Byte);

```

说明：计算 Dimen 维方阵的行列式的值。Data 为矩阵数据。Det 为行列式的值。Error 为错误代码。

求逆矩阵

```

语法：Procedure Inverse(Dimen; Integer; Data; TNmatrix;
                          Var Inv; TNmatrix; Var Error ; Byte);

```

说明：求 Dimen 维方阵的逆矩阵。Data 为矩阵数据。Inv 为求出的逆矩阵，Error 为错误代码。

用高斯消去法解线性方程

```

语法：procedure Gauss—Elimination(Dimen; Integer;
                                      Coefficients; TNmatrix;
                                      Constants; TNvector;
                                      Var Solution; TNvector;
                                      Var Error ; Byte);

```

说明：用高斯消去法解线性方程。Dimen 为系数矩阵的维数，Coefficients 为系数矩阵。Constants 为常数项，Solution 方程组的解。Error 为错误代码。

直接分解法解线性方程组

```

语法：procedure Lu—Decompose(Dimen; Integer;
                               Coefficients; TNmatrix;
                               Var Decomp ; TNmatrix;
                               Var Permute ; TNmatrix;
                               Var Error ; Byte);

```

说明：首先用 LU—Decompose 将矩阵分解为一个上三角矩阵和一个下三角矩阵，然后用



LU—Solve 解线性方程组。这两个都是数据库工具箱中的过程。Dimen 系数矩阵的维数。Coefficients 为系数矩阵。Decomp 为系数矩阵的分解。Permute 为置换矩阵。Error 为错误代码。

语法: Procedure Lu—SoLve(Dimen; Integer, Var Decomp;  
TNmatrix; Constants; TNvector;  
Var Permute TNMatrix;  
Var Solution; TNvector;  
Var Error; Byte);

说明: Constants 为常数项, Solution 为方程组的解。其它同上。

幂元法求实矩阵的主特征值和特征向量

语法: procedure Power(Dimen; Integer; Var Mat;  
TNmatrix; Var GuessVector;  
TNvector; MaxIter; Integer;  
Tolerance; Real; Var EigenValue;  
Real; Var EigenVector; TNvector;  
Var Iter; Integer; Var Error; Byte);

说明: 求矩阵 Mat 的主特征值和特征向量。

Dimen 矩阵的维数, GuessVector 特征向量的初值, MaxIter 最大迭代次数, Tolerance 解的精度。EigenValue 求出的主特征值, EigenVector 主特征向量。Iter 迭代次数, Error 错误代码。

Jacobi 法求实对称矩阵的完全特征系

语法: Procedure Jacobi(Dimen; Integer; Mat; TNmatrix;  
MaxIter; Integer; Tolerance; Real;  
Var Eigenvalues; TNvector;  
Var Eigenvectors; TNvector;  
Var Error; Byte);

说明: 求对称矩阵 Mat 的完全特征系。Dimen 矩阵的维, MaxIter 最大迭代次数, Tolerance 解的精度。Eigenvalues 求出的特征值, Eigenvectors 特征向量, Iter 迭代次数, Error 为错误代码。

求一阶常微分方程的初值问题的解

语法: procedure InitialCondition1Storder(LowerLimit; Real;  
UpperLimit; Real;  
XInitial; Real;  
NumReturn; Integer;  
NumInterval; Integer;  
Var TValues; TNvector;  
Var XValues; TNvector;  
Var Error; Byte);

说明: 用龙格—库塔方法求具有给定初值条件的一阶常微分方程的近似解。LowerLimit 为区间的下限, NumReturn 返回的 (X, Y) 对的数目, NumIntervals 区间数。TValues 为 t 的值, XValues 为 t 处 X 的近似值, Error 错误代码。

二阶常微分方程初值问题。

语法: procedure InitialCond2Order(LowerLimit; Real;  
UpperLimit; Real;  
Initialvalue; Real;  
InitialDeriv; Real;  
NumReturn; Integer;  
NumIntervals; Integer;  
Var TValues; TNvector;  
Var XDerivValues; Invector;  
Var Error; Byte);

说明: 用龙格-库塔法求二阶常微分方程的初值问题的解。LowerLimit 为区间下限, UpperLimit 为区间上限。Initialvalue 下限处 X 的初值, InitialDeriv 下限处 X 的导数。NumReturn 返回的 (t, x) 的个数, NumIntervals 区间数。TValues 为 t 的值。XValues 为 X 的值, XDerivValues 为 X 的一阶导数。Error 为错误代码。

二阶常微分方程的边值问题。

语法: Procedure Shooting(LowerLimit; Real; UpperLimit;  
Real; LowerInitial; Real;  
UpperInitial; Real; Initialslope;  
Real; NumReturn; Integer;  
Tolerance; Real; MaxIter;  
Integer; NumIntervals; Integer;  
Var Iter; Integer; Var Xvalues;  
TNvector; Var Yvalues; TNvector;  
Var YDerivvalues; TNvector;  
Var Error; Byte);

说明: 求给定边值条件的二阶常微分方程的近似解。LowerLimit 和 UpperLimit 分别为区间的上, 下限, LowerInitial 和 UpperInitial 分别为 Y 在上, 下限处的值。Initialslope 为下限处的近似斜率。NumReturn 为返回的 X, Y 和 Y 的一阶导数值的个数。tolerance 为解的精度, MaxIter 为最大迭代次数, NumIntervals 为区间个数。Iter 为迭代次数。XValues 为 X 的值, YValues 为对应 X 的 Y 值。YderivValues 为 Y 值对应的一阶导数值。Error 为错误代码。

最小二乘近似

语法: procedure LeastSquares(NumPoints; Integer;  
Var Xdata; TNcolumnVector;  
Var YData; TNcolumnVector;  
Var NumTerms; Integer;  
Var Solution; TNcolumnVector;  
Var StandardDeviation; Real;  
Var Error; Byte);

说明: 求一组点 X, Y 的最小二乘近似。NumPoint 为点的个数, XData 为 X 坐标, YData 为 Y 坐标, NumTerms 为最小二乘近似中项的个数。Solution 为基向量的系数, StandardDevi-

ation 为标准方差,Error 为错误代码。

### 快速傅里叶变换

语法: Procedure FFT(NumberOfBits;Byte;NumPoints;  
Integer;Inverse;Boolean;  
Var XReal ;TNvectorPTR;  
Var XImag ;TNvecTorptr;  
Var SinTable;TNncetorptr;  
Var CosTable;TNvcetorptr);

说明: 对数据作快速傅里叶变换.NumberofBits 为数据点的幂, 为2 或 4.NumPoints 数据点的个数.Inverse 指出变换方向, False 向前变换, TURE 后向变换.XReal 数据的实部, XImag 为虚部.SinTable 正弦值表, Costable 为余弦值表。

## 第九章 编程技术

在编写程序的过程中,有一些方法是要经常使用的。掌握这些方法可帮助程序员更好更快地编写程序。有些方法可以用 Turbo Pascal 的标准过程和函数来完成。有些方法可以用标准化的高效专门算法来完成。下面介绍其中一些有关内容。

### § 9.1 字符串

在 Turbo Pascal 中,字符串是一种常用的数据结构。对于字符串,可以用两种不同的方式进行处理:一是对其中的每个元素单独进行处理,二是将字符串作为一个整体进行处理。在对字符串进行处理时,可以使用 Turbo Pascal 提供的标准字符串过程和函数。下面介绍标准字符串过程和函数。

#### Chr

这个函数接受一个整数,返回对应的 ASCII 码。这不是一个字符串过程,但在字符串处理过程中,经常要用到这个函数,特别是用于处理字符串中的特殊字符。例如 ASCII 码中的前32个字符,这些字符常常是控制码,用通常的方法不能显示和打印。

#### Uppcase

这个函数将小写字母变为相应的大写字母。如果不是小写字母,则不变。

#### Concat

将几个字符串连接为一个字符串。Concat 接收若干个字符串作为参数,把它们作为一个字符串返回。这个函数与运算符+的效果相同。程序员更愿意使用+运算符。这主要是简单和直观。当函数 Concat 和运算符+产生的字符串长度大于字符串允许的最大长度时,结果字符串将被截断。

#### Copy

这个函数从一个字符串中截取一个子串。Copy 的形式为:

Copy(S,P,L);

其中 S 是字符串, P 是截取子串的起始位置。L 是子字符串的长度,在字符串的尾部截取时,如果 S 中 P 的后的字符个数小于 L,可能截取不到 L 个字符。但这并不产生错误信息。

#### Delete

这个过程用于从一个字符串中删除一个子串。Delete 的形式如 Copy 相似。

Delete(S,P,L);

参数 S, P, L 的含义与 Copy 中的相似,这条语句是从 S 字符串中删除从 P 开始的 L 个字符。

#### Insert

这个过程是把一个字符串加到另一个字符串中。Insert 的形式如下:

Insert(S1,S2,P);

其中 S1 和 S2 是字符串, P 是整数。这条语句将字符串 S1 插到 S2 中第 P 个位置中。

Length

这个函数返回字符串中字符的个数。例如 S 为 'This is a string.', 则 Length(s) 为 20, 应注意的是字符串后的空格也是字符串的一部分。

Pos

这个函数返回一个子字符串在字符串中的位置, 其形式为:

Pos(s1, s2);

其中 S1 和 S2 为字符串, 这个函数返回 S1 在 S2 中的位置。若 S1 不在 S2 中, 则返回 0。

Str

这个函数将一个数字串转化为字符串, 其形式为:

Str(N, S);

其中 N 为数字, S 为字符串, 将 N 变为字符串放入 S 中。

Val

这个函数的作用与 Str 的作用相反, 将一个字符串转换为数字。其形式为:

Val(S, N, Code);

其中 S 是字符串, N 为数字, 如果转换成功, Code 为 0, 否则 Code 为非零值。对于这个函数输入的字符串 S 必须满足下列条件:

1. S 必须是一个整数, 实数和科学记数法形式的数。
2. S 中不得有字母或其它非数字字符, 科学表示法中的 "E" 除外。
3. S 不能有后接的空格, 前面的空格可以接受。

当把一个有效实数的字符串转换为整数时, 会产生一个错误信息。因此把所有的数字转换为实数是安全的, 然后用 Round 或 Trunc 转换为整数。

在上面介绍的 Turbo Pascal 标准过程和函数中, 可以处理很多字符串问题。但有的时候需要一个一个地处理字符串的每个字符。例如将一个字符串中的所有字符变为大写字母。在对字符串中的单个字符进行处理时, 可以从字符串的第一个字符开始, 向后处理。字符串的第一个字节是字符串的长度。例如若 S 是字符串, 则  $j := \text{Ord}(S[0])$ ; 将字符串的长度赋给 j。同样,  $j := \text{Length}(S)$ ; 也是将字符串的长度赋给 j。这两条语句的功能是相同的。

利用字符串的第一个字节, 可以增加和缩短一个字符串的长度, 但不必改动整个字符串。例如:

S := 'ABCDEFGH';

s[0] := Char(3);

WriteLn(S);

在第一条语句中, 把 'ABCDEFGH' 赋给 S 时, 同时也将其长度字节设置 ASCII 码的 7。在第二条语句中, 把长度字节变为 ASCII 码的 3。因为第一个字节也看作字符, 所以要用 Chr 函数。这时, S 的内容虽然没有变, 但其长度却变成了 3。因此在第二条语句中, 给出结果为 'ABC'。

同样, 也可以直接修改字符串的字符, 而不改变字符串的长度字节。例如:

S := 'ABC';

s[4] := 'D'; S[4] := 'E' / WriteLn(s);

第一条语句把 'ABC' 赋给 S, 同时将长度设置为 3。接下来的两条语句修改字符串的第 4

和第5个字符,但是这不影响字符串的长度,所以在第四条试句中 WriteLn(s)的输出结果仍为 'ABC',而不是 'ABCDE'。

直接修改字符串和其长度字节是很有用的方法。有时需要一串相同的字符,但在程序的不同地方,长度不同。这时,可以通过修改长度字节,来满足不同的需要。例如:

```
FillChar(S,Size-Of(s),205);
```

将 s 字符串的255个字符都设置为 ASCII 码的205。当需要80个字符串时,可用语句:

```
s[0]:=Chr(80);
```

这时可以使 S 字符串只有80个字符。

下面介绍用字符串如何解决编程问题,首先介绍在一个字符串中寻找,删除和替换一个字母组合。下面看一个程序:

```
Program SearchAndReplace;
```

```
Uses CRT;
```

```
Var
```

```
    BigString : String[255];
```

```
    FindString;
```

```
    ReplaceString : String[20];
```

```
    i : Integer;
```

```
Begin
```

```
    ClrScr;
```

```
    FindString := 'Steve';
```

```
    ReplaceString := 'Jong';
```

```
    BigString :=
```

```
    'Tell Steve to pay me the five dollars he owes me.';
```

```
    WriteLn(BigString);
```

```
    i := Pos(FindString,BigString);
```

```
    Delete(BigString,i,Length(FindString));
```

```
    Insert(ReplaceString,BigString,i);
```

```
    WriteLn(BigString);
```

```
    WriteLn
```

```
    Write('Press ENTER...');
```

```
    ReadLn;
```

```
End.
```

在这段程序中共用了4个标准字符串过程: Pos, Delete, Insert 和 Length。这段程序中首先把字符串 'Tell Steve to pay me the five dollars he Owes me'. 赋给 Bigstring,然后在这个字符串中确定要替换的子串 'Steve' 在 Bigstring 中的位置。即使用语句。

```
i:=Pos(FindString, Bigstring);
```

然后用 Delete 过程删除这个子串。然后把替换子串用 Insert 插入到字符串中。

这种方法可以用于处理更复杂的问题。在字符串中要出现名字的地方用@字符代替。然后在需要时，用名字替换@字符。如 'Hello,@', 用 John 代替@后，字符串即为 'Hello,John'。但是使用这种方法有一个缺点，即在字符串中不能出现@字符。例如："This is the @ Character, @'。想保留第一个@，而把第二个@ 变成名字，但结果却不是这样。因此，最好用一个特殊的非打印字符代替@ 字符。

字符串还可用于数字的输入，然后将字符串转换为数字。如果直接从键盘输入数字，当出现错误时，如键入了无效数字或带空格的数字，Turbo Pascal 将产生运行错误并使程序停止。如果在字符串转换时出现错误，可以要求重新输入。例如下面这段程序：

```
Program EnterNumber;
Uses CRT;
Var
    Age, Code : Integer;
    AgeString : String[10];
Begin
    ClrScr;
    Repeat
        Write('Enter your age: ');
        ReadLn(AgeString);
        Val(AgeString, Age, Code);
        If Code <> 0 Then
            Write(^g); (* Make the computer beep *)
    Until Code = 0;
    WriteLn('Your age is : ', Age);
    WriteLn;
    Write('Press ENTER... ');
    ReadLn;
End.
```

在这段程序中，要求输入年龄。这时年龄是作为字符串输入的，然后用 Val 函数将这个字符串转换为数字，如果转换失败，则响铃，并要求重新输入，直到转换成功。这样就避免了由于输入的数字错误而使程序停止执行。

由于数字后面的空格会使转换失败，例如 '101 ' 不能转换为一数字，必须把数字后面的空格去掉。这可以通过一个循环来实现。如：

```
While(S[Length[s]]='')Do
    Delete(S, Length(s), 1);
End;
```

## § 9.2 递归

递归是一种复杂的编程技术。所谓递归就是一个过程调用其自身。对于这种技术是比较难

于把握的。有些问题可以使用递归和非递归两种方法实现。至为使用哪种方法，要由几个方面来决定。我们先看一个简单的求整数  $N$  的阶乘的例子。使用非递归方法的程序如下：

```
r:=1;
For i:=2 To n Do
  r:=r*i;
```

这种方法是直接计算  $N$  的阶乘。如果使用递归方法2，见下面的例子：?Function Factorial  
(n:Integer);real;

```
begin
  If n=0 Then Factorial:=1
  Else
    factorial:=n * Factorial(n-1);
  End;
```

递归方法是使  $n$  与  $n-1$  的阶乘相乘。反复这样做直到  $n=0$  为止，虽然递归方法漂亮而且精致，但非递归法更易于理解和编码。使用递归方法的一个主要问题是，每次调自身时都要设置必要的堆栈空间，以放置临时变量。这不但占用内存空间，减低程序的执行速度，而且，当递归占用的空间太大时，可能使程序崩溃。

但是另一方面，有一些算法更适合递归结构。如果使用非递归结构会显得不自然而且不合理。下面看一个例子。

```
Program Calculator;
Uses CRT;
Type
  MaxCompStr = String[255];
Var
  i : Integer;
  Formula : String[80];
  Result : Real;
  Error : Boolean;
```

```
( * * * * * )
```

```
Function Compute—Formula(Var p : Integer;
                           Strg : MaxCompStr;
                           Var Error : Boolean) : Real;
Var
  r : Real;
  i,
  BreakPoint : Integer;
  Ch : Char;
```



( \* \* \* \* \* )

```
Procedure Eval(Var Formula ; MaxCompStr;  
               Var Value ; Real;  
               Var BreakPoint ; Integer);
```

```
Const  
  Numbers ; Set Of Char = ['0'..'9','.','];  
Var  
  p,i ; Integer;  
  Ch ; Char;
```

( \* \* \* \* \* )

```
Procedure NextP;  
Begin  
  Repeat  
    p := p+1  
  If p <= Length(Formula) Then  
    Ch := Formula[p]  
  Else  
    Ch := #13;  
  Until (Ch <> ' ');  
End;
```

( \* \* \* \* \* )

```
Function Expr ; Real;  
Var  
  E ; Real;  
  Operator ; Char;
```

( \* \* \* \* \* )

```
Function SmpExpr ; Real;  
Var  
  S ; Real;  
  Operator ; Char;
```

( \* \* \* \* \* )

Function Term : Real;

Var

T : Real;

( \* \* \* \* \* )

Function S—Fact : Real;

( \* \* \* \* \* )

Function Fct : Real;

Var

fn : String[20];

l,start : Integer;

F : Real;

( \* \* \* \* \* )

Procedure process—as—number;

Var

code : Integer;

Begin

Start := p;

Repeat

Nextp

Until Not(Ch In Numbers);

If Ch = '.' Then

Repeat

NextP

Until Not(Ch In Numbers);

If Ch = 'E' Then

Begin

NextP;

Repeat

NextP;

Until Not(Ch In Numbers);

End;

Val(Copy(Formula, Sart, p—Start), F, code);

End;

( \* \* \* \* \* )

Procedure process—as—nes—Expr;

Begin

NextP;

F := Expr;

If Ch = ')' Then

NextP

Else

BreakPoint := p;

End;

( \* \* \* \* \* )

Procedure Process—as—standard—Function;

( \* \* \* \* \* )

Function Fact(I ; Integer) ; Real;

Begin

If I > 0 Then

Fact := I \* Fact(I-1)

Else

Fact := 1;

End;

( \* \* \* \* \* )

Begin

If Copy(Formula, p, 3) = 'ABS' Then

Beign

P := P + 2;

NextP;

F := Fct;

f := Abs(f);

End

Else If Copy(Formula, p, 4) = 'SQRT' Then Begin

P := P + 3;

NextP; F := Fct;

f := Sqrt(f);

End

Else If Copy(Formula, p, 3) = 'SQR' Then

```

    Begin
    P := P+2;
    NextP;
    F := Fct;
    f := Sqr(f);
    End
Else If Copy(Formula, p, 3) = 'SIN' Then
    Begin
    P := P + 2
    NextP;
    F := Fct;
    f := Sin(f);
    End
Else If Copy(Formula, p, 3) = 'COS' Then
    Begin
    P := P + 2;
    NextP;
    F := Dct;
    f := Cos(f);
    End
Else If Copy(formul, p, 6) = 'ARTAN' Then
    Begin
    P := + 5;
    NextP;
    F := Fct;
    f := ArcTan(f);
    End
Else If Copy(Formula, p, 2) = 'LN' Then
    Begin
    P := p + 1;
    NextP;
    F := Fct;
    f := Ln(f);
else if Copy(Formula, p, 3) = 'EXP 'Then
    Begin
    P := P + 2 ;
    NextP;
    F := Fct;
    f := Exp(f);
    End

```

Else if Copy(formula, p, 4) = 'ACT' Then

Begin P := P + 3;

NextP;

F := Fct;

f := fact(Trunc(f));

end

else

Begin

BreakPoint := p;

End;

End;

( \* \* \* \* \* )

Begin ( \* process--as--standard--Function \* )

If (Ch In Numbers) Then

Process--as--number

Else If (Ch = '(') Then

Process--as--new--Expr

Else

process--as--standard--Function;

Fct := F;

End; ( \* process--as--standard--Function \* )

( \* \* \* \* \* )

Begin

If Ch = '-' Then

Begin

NextP;

S--Fact := --Fct;

End

Else

S--Fact := Fct;

End;

( \* \* \* \* \* )

```

BeginT := S--Fact;
While Ch = '^' Do
  Begin
    NextP;
    t := Exp(Ln(t) * S--Fact)
  End;
Term := t;
End;

( * * * * * )
Begin
s := term;
While Ch In ['*', '/'] Do
  Begin
    Oerator := Ch;
    NextP;
    Case Operator Of
      '*' : s := s * term;

      '/' : s := s / term;
    End;
  End;
SmplExpr := s;
End;

( * * * * * )
Begin
E := SmplExpr;
While Ch In ['+', '-'] Do
  Begin
    Operator := Ch;
    NextP;
    Case Operator Of
      '+' : e := e + SmplExpr;
      '-' : E := E -- SmplExpr;
    End;
  End;
Expr := E;
End;

```

( \* \* \* \* \* )

```
Begin
For i := 1 To Length(Formula) Do
    Formula[i] := Ucase(Formula[i]);
If Formula[i] = '.' Then Formula := '0'+Formula;
If Formula[i] = '+' Then Delete(Formula, i, 1);
P := 0;
NextP;
Value := Expr;
```

```
If Ch = #13 Then
    Error := False
Else
    Error := True;
```

```
BreakPoint := p;End;
```

( \* \* \* \* \* )

```
Begin
Eval(Strg, r, p);
Compute—Formula := r;End;
```

( \* \* \* \* \* )

```
Begin
ClrScr;
Repeat
    Write('Enter Formula: ');
    Read(Formula);
    If Formula <> '' Then
        Begin
            Result := Compute—Formula(p, Formula, error);
            If Error Then
                Begin
                    WriteLn;
                    WriteLn('Error!');
                    WriteL(Formula);
                    For i := 1 To p-1 Do Write(' ');
                    WriteLn('^ ');
                End
            End
        End
    End
Until Error = False;
```

```

    End
Else
    WriteLn(' = ', Result;02);
End;
ReadLn;

```

```
Until Formula = '';End.
```

这段程序要求输入一个方程，放入一个字符串中。然后将字符串传给函数 Compute—Formula，这个函数通过一系列递归调用来计算方程。如果成功，将结果返回程序，否则将 Error 变量设置为 TRUE，P 则指向出错的位置。这段过程也可以用非迭代方式编写，但是对这种算法，迭代算法更适用。

### § 9.3 DOS 设备

计算机的输入输出主要是通过键盘，显示器，打印机和磁盘等实现的。在 Turbo Pascal 中这些设备都可以作为文件来处理。这样可以统一处理所有的输入输出。这些设备称作 DOS 设备。还有一些设备是 DOS 不支持的。称为非 DOS 设备，如鼠标器等。非 DOS 设备要用自己的设备驱动程序。

标准的输出输入设备都为 CON，但对输入设备，CON 是键盘，输出设备为视频显示。使用标准设备可以象文件一样。例如

```

Assign(f, 'CON');
Rewrite(f);
WriteLn(f'Output to CON');

```

将标准设备赋以的文件指针后。可以象写文件一样，向视频显示设备输出数据。同样，将标准设备赋以的文件指针后，可以象读文件一样，从键盘输入数据。例如：

```

Assign(f, 'CON');
Reset(f);
ReadLn(f,s);

```

同样，DOS 支持各种打印设备，如：PRN1, LPT1, LPT2, LPT3, LPT1 和 PRN 是同一设备。由于大部分计算机只配备一台打印机，只使用 LPT1 或 PRN。打印机只用于输出，如果对打印机使用 Reset 命令，Turbo Pascal 将产生文件结果符号。

Turbo Pascal 还提供了另外一种使用打印机的方法：Printer 单元。这个单元说明一个文本文件名 LST，使用这个单元输出将指向打印机。下面的例子说明如何使用 LST 设备：

```

Program Prt;
uses Printer;
Begin
WriteLn(LST, 'ABCD');
End.

```

大部分计算机都有串行接口，用于打印机，调制解调器，局域网和其它通讯设备。Turbo Pascal 支持两个 DOS 串行设备 COM1 和 COM2。此外。还支持一个辅助设备 AUX，通常这个



设备与 COM1 相同。虽然使用这些设备可以方便地从串行接口发送或接受数据。通讯程序一般直接使用串行接口，而不使用 DOS 设备。

Turbo Pascal 还支援一种特殊的设备：NUL 设备。向这个设备发送的一切数据都予以忽略。当编写一个输出程序，而不想实际发送任何东西时，可以使用这个设备。

## § 9.4 合并

合并是计算机程序设计中经常用到的设计方法。所谓合并是将两个排好序了的文件合成一个更大的排序文件。可以将一个文件接在另一个文件之后，然后再对文件进行一次总排序。但这种方法要比合并法需要更多的时间。

合并过程程序开始的先入两个文件中都读出第一个记录，然后进入一循环。在循环中比较两个记录，将值小的记录放入新文件中，再从输入文件中读出下一个记录。直到其中的一个文件读完。然后把剩下的那个文件记录都写入新文件。

合并过程说起来很简单。但要实现起来并非那么简单。主要问题是确是什么时候从一个输入文件中读出一个记录，及输入文件是否已空。下面看一段程序。

```
Program MergeTst;
Uses CRT;
Type
  Str80 = String[80];

Var
  File1,
  File2,
  File3, Str80;

( * * * * * )

Procedure Merge(Fname1, Fname2, Fname3 : Str80);
Var
  ok1, ok2 : Boolean;
  f1, f2, f3 : Text;
  i1, i2 : Integer;

Function GetItem1(Var i : Integer) : Boolean;
Begin
  If Not Eof(f1) Then
    Begin
      ReadLn(f1, i);
```

```

    GetItem1 := True;
End
Else
    GetItem1 := False;
End;

```

( \* \* \* \* \* )

```

Function GetItem2(Var i : Integer) : Boolean;
Begin
If Not Eof(f2) Then
    Begin
        ReadLn(f2,i);
        GetItem2 := True;
    End
Else
    GetItem2 := False;
End;

```

( \* \* \* \* \* )

```

Begin
Assign(f1,Fname1);Reset(f1);
Assign(f2,Fname2);
Reset(f2);
Assign(f3,Fname3);
Rewrite(f3);

```

```

ok1 := GetItem1(i1);
ok2 := GetItem2(i2);

```

```

While ok1 Or ok2 Do
    Begin
        (* If ok1 is true, then a record from File 1 is present. *)
        (* If ok2 is true, then a record from File 2 is present. *)
    End

```

```

If ok1 And ok2 Then    (* records are present *)
    Begin              (* from both files.      *)
        If i1 < i2 Then
            Begin

```

```

        WriteLn(f3,i1);
        ok1 := GetItem1(i1);
        End
Else
    Begin
        WriteLn(f3,i2);
        ok2 := GetItem2(i2);
        End;
    End
Else If ok1 Then      ( * a record is present from * )
    Begin              ( * the first file only.      * )
        WriteLn(f3,i1);
        ok1 := GetItem1(i1);
        End ( * a record is present from * )
    Else If ok2 Then   ( * the second file only.      * )
        Begin
            WriteLn(f3,i2);
            ok2 := GetItem2(i2);
            End;
        End
    Close(f1);
    Close(f2);
    Close(f3);
    End;

( * * * * * )

BeginClrScr;
Write('Enter Name of first file: ');

ReadLn(File1);Write('Enter name of second file: ');
ReadLn(File2);
Write('Enter Name of merged file: ');
Merge(file1,file2,file3);
End.

```

这个程序首先接受三个文件名: File1, File2 和 File3。File1和 File2是输入文件。File3是合并文件。然后将这三个文件名传给排序过程 Merge。这个过程用 GetItem1和 GetItem2两个函数分别从两个文件中读出一个记录。若读出成功,则返回 TRUE。否则,若达到文件尾,则返

回 FALSE。两个变量 OK1 和 OK2 分别保存这两个函数的返回值。用下面的语句

```
while OK1 Or OK2 Do
```

来控制循环。

只要还有一个文件没有结束，就继续执行循环体，直到两个文件都到达文件尾。在这个循环中有三个分枝。当两个文件都有记录时执行第一个分枝。比较两个记录，将值小的记录放入合并文件，并读出下一个记录。

当有一个文件读到文件结束标记时，执行下两个分枝。从未读到文件尾的文件中读出记录，放入合并文件中。直到这个文件也遇到文件结束标志。然后退出循环，关闭所有文件，结束程序。

## § 9.5 排序

排序方法有很多种，现在常用的有三种：气泡法，Shell 法和快速法。气泡排序法最容易编写，但执行速度极慢。Shell 排序法的速度适中，占用的内存比气泡法多。快速排序法速度最快，但要占用相当大的堆栈空间用于递归。了解这三种方法，明白其中的优劣之处对程序设计很有帮助。下面分别介绍这三种排序方法。

### 气泡排序法

排序的一般方法是将数组中的一个元素与另一个元素比较，如果不是顺序的，则交换两个元素在数组中的位置。这三种排序法的主要区别在于比较数组元素的方法不同。采用不同的比较方法对排序有极大影响。例如，气泡排序法只比较相邻的元素，因此可能需要几千万次比较才能排好一个数组。而在同样的情况下，使用快速排序法只需几千次比较就可把数组排好。

气泡排序法很简单：从数组的后端开始，将每个元素与前而的元素比较，如果顺序不对则交换两个元素。一直进行到数组的头。这样，在第一遍扫描后，最小的元素将象气泡一桩浮到明面。第二遍扫描后，次小元素将浮到第二位。这样直到数组中的每个元素都扫描一遍为止。下面这段程序给出了 Turbo Pascal 是如何对  $n$  个元素的数组进行排序的。

```
For i:=2 to n Do  
  For j:=n Down To i Do  
    If a[j-1]>a[j] Then  
      Switch(a[j],a[j-1]);
```

这段程序的内层循环执行每行中的比较。它是由尾部向前比较，由于  $i$  每次扫描后都加 1，（扫描行一次比一次短。外层循环即控制每次扫描的次数。在比较部分只有一条语句。因此气泡算法是很紧凑的。

气泡排序法每扫描一次，则下一个最小元素放到数组中的适当位置，而较大的数则向后移。气泡排序法的缺点在于只比较相邻的数组元素。

### Shell 排序法

这种排序法比气泡排序法的效率要高得多。该算法的主要优点在于对元素最终位置的估计。它将元素先近似地置位，然后再准确置位。在 Shell 排序法中的重要概念是“间距 (Gap)”。它是指两个比较元素之间的距离。在对整个数组扫描一遍之后，相隔一定间距的元素已处于排好序的状态。当间距缩小到 1 后，再描一次即可使整个数组排好序。扫描间距的初始值是任

意的,通常用数组元素个数的一半,即  $n \text{ DIV } 2$ 。

shell 排序法有许多不同的种类。下面介绍其中一种,这种 Shell 排序法的效率很高,只需很少几次扫描。下面是这种排序法的基本代码:

```
Gap:=nDIV2;
While (Gap>0) Do
Begin
  For i:=(gap+1) To n Do
    Begin
      j:=i-Gap;
      While(j>0) Do
        Begin
          k:=j+Gap;
          If(a[j]<=a[k]) Then
            j:=0,
          Else
            Switch (a[j],a[k]);
            j:=j-Gap;
          End;
        End;
      Gap:=Gap DIV 2,
    End;
```

在这段程序中,首先将间距设置为  $n \text{ DIV } 2$ 。外循环由间距 Gap 控制,每扫描一次后,间距除2,直到 Gap 为0。每次扫描中用三个变量来决定要比较的元素: i, j 和 k。i 指向远元素, j 指向近元素。k 在比较之前为  $k:=j+\text{Gap}$ , 实际为 i。在比较中用 k 代替 i 是因为有时要向前比较。在向前比较时, i 每次减少 Gap, k 也将随之减小。而 j 是控制内循环的,这时不应减少其值,即 i 控制算法流程, k 在必要时回溯。这一逻辑上的技巧比起最简单的 Shell 算法效率提高了三倍。

#### 快速排序法

虽然 Shell 排序法效率很高,但快速排序法要比它快二倍。这种算法广泛应用于各种通用排序程序。

这种排序法速度快的一个原因是,它与人们通常的排序方法很相似。首先将元素分成几份,然后继续分成更小的份。逐渐将数组排好序。分份的方法是先估计数组元素的中间值。中间值的具体值无关紧要。但是越接近实际,排序就越快。通常是将首尾两数的中值作为数组元素中间值。把所有个中间值小的元素放到前面,比中间值大的元素放到后面。然后对中间值两边的数组再进行排序。随着划分部分越来越小,数组就逐渐排好,下面一段程序包括快速排序法。

```
Program QuickTest;
Uses CT;
Type
  Int—Arr = Array [1..10] Of Integer;
```

```

Var
  InFile ; Text;
  i ; Integer;
  a ; Int—Arr;

```

```

( * * * * * )

```

```

Procedure quick(Var item ; Int—Arr; count ; integer);

```

```

( * * * * * )

```

```

Procedure PartialSorr(left, right ; Integer;
                      Var a; Int—Arr);

```

```

Var
  ii,
  ll,rl,
  i,j,k ; Integer;

```

```

( * * * * * )

```

```

Procedure Switch(Var a,b ; Integer);Var

```

```

  c ; Integer;
Begin
  If a <> b Then
    Begin
      c := a;
      a := b;
      b := c;
    End;

```

```

( * * * * * )

```

```

Begin
  k := (a[left] + a[right]) Div 2;
  i := left;
  j := right;
  Repeat

    While a[i] < k Do
      Inc(i,1);

```

```

While k < a[] Do
    Dec(j,1);

    If i <= j Then
        Begin
            Switch(a[i],a[j]);
            Inc(i,1);
            Dec(j,1);
            End;
Until i > j;

If left < j Then
    PartialSort(left,j,a);
If i < rigth Then
    PartialSort(i,rigth,a);
End;

```

( \* \* \* \* \* )

```

Beking
PartialSort(l,count,item);
End;

```

begin

```

ClrScer;
a[1] := 86;
a[2] := 3;
a[3] := 10;
a[4] := 23;
a[5] := 12;
a[6] := 67;
a[7] := 59;
a[8] := 47;
a[9] := 31;
a[10] := 24;

```

```

For i := 1 To 10 Do
    Write(a[i]:4);

```

```

WriteLn;

Quick(a,10);

For i := 1 To 10 Do
    Write(a[i];4);
    writeLn;
Write('Press ENTER...');
ReadLn;
End.

```

程序首先将数组和元素的个数传递给 Quick 过程。Quick 再将数组和上、下限传给子过程 Partialsort。开始时，下限为1，上限为数组的元素个数。Partialsort 求出中间值，并进行排序。再将新的上、下限递归调用自身，直至不能再分。这种方法由于要递归调用，要占用相当大的内存。

与 Shell 排序法和快速排序法比较，气泡法排序法算不上一种好方法，它需要多用几倍到几十倍的时间。Shell 排序法与快速排序法比较差别也很明显。它要多用一倍以上的时间。但是快速排序法也有缺点，它要占用大量的堆栈空间。若需要考虑堆栈空间，应采用 Shell 排序法。

## 9.6 搜索

搜索是指在许多数据中寻或特定的数据。搜索可以用几种不同的方法取得。这里介绍两种搜索方法：顺序搜索和折半搜索。

### 顺序搜索

顺序搜索就是从数组的第一个元开始，将搜索值与数组元素一个一个地比较，直到找到匹配的值。搜索过程需要知道三个参数：要搜索的值，数组及数组中元素个数。下面是顺序搜索的代码。

```

For i := 1 To n Do
    If x = a[i] Then
        Begin
            Seqsearch := i;
            Exit;
        End
    Seqsearch := 0;

```

在这段程序中，X 是要搜索的值，a 是数组，N 是数组中的元素个数，若找到匹配的值，则令 Seqsearch 为匹配元素的序号。并退出。若未找到匹配的元素，则令 Seqsearch 为0。

由于这种方法是依次对数组元素进行处理。因此数组是否排好序无所谓。

### 折半搜索

折半搜索要求数组必须排好序。开始时，先将搜索值与中间值的元素比较。如搜索值非于中间元素，则搜索后半部分。否则搜索前半部分。再与剩下部分的中间元素比较。直至找到



匹配元素,或只剩下一个元素,但不匹配为此。

折半搜索比顺序搜索有效的多。例如,有100个元素的数组,用顺序搜索平均要比较50次,而折半搜索最多只需比较7次,数组元素越多,折半搜索的优越性越明显,下面是折半搜索的基本代码:

```
Low:=1;
High:=n;
While High>=Low Do
  Begin
    Mid:=Trunc((High+Low) Div 2);
    If x>a[Mid] Then
      Low:=Mid+1;
    Else If x<a[Mid] Then
      High:=Mid-1;
    Else
      High:=-1;
    End;
    If High=-1 Then
      Bsearch:=Mid
    Else
      Bsearch:=0;
    End;
```

Low 和 High 是搜索的区间。开始时, Low 为1, High 为 n, 其中 n 为数组中的元素个数。每比较一次, 若中间元素不匹配, 区间就减少一半, 即 High 变为 Mid-1 或 Low 变为 Mid+1。这样两个值越来越接近。若 Low 大于 High, 则说明数组中没有匹配的元素, Bsearch 设置为 0。如果对某个 Mid, a[mid] 等于搜索值, 令 High 就-1。使循环终止, 并使 Bsearch 等于匹配元素的位置 Mid。

## 第十章 视频：文本显示与图形

视频特性对于个人计算机上运行的程序来说是极为重要的。首先是几乎每个程序都会有视频输出，好的视频输出会极其准确地表达程序的思想，有助于对程序的理解。其次对于许多计算机用户来说，显示器和键盘就是计算机，他们常常根据视频显示的质量来判断程序的优劣。在许多时候，一仅仅个有吸引力的视频输出，就可能成为程序获得成功的真正原因。

个人计算机是具备产生优质视频输出的潜在能力的，但是开发这种能力，对视频显示器进行合适的编程，都是一种挑战。本章就将讨论视频显示器的能力以及如何使用 Turbo Pascal 来控制它们。

### § 10.1 视频显示的基本概念

在深入了解 Turbo Pascal 对显示效果的具体控制手段之前，有必要了解视频显示的一些基本概念。

#### 1. 文本与图形

个人计算机有两种很不相同的视频显示方式：文本方式与图形方式。在文本方式下，计算机可以显示 256 个不同的 ASCII 字符，这些字符的编码及显示形状在个人计算机上基本上是标准的。这些字符永久地锁定存贮于计算机中，因此不必担心 PC 机不知道如何描画它们。在图形方式下，计算机不仅可以显示字符，也可以显示任何我们能够描绘的图案、图表，包括多字体的文本。使用计算机的图形能力是利用视频显示特性的最高境界，它比使用文本方式要难得多，因为我们每次都需要告诉计算机所显示的图案该如何画。我们首先要会画线和画字符，然后还得会把它们弄成合适的大小放到合适的位置。更麻烦的是，你还必须面对不下于十种目前可用的图形适配器，以及它们构造的数十种不同的图形方式。而对于文本方式来说，不同适配器之间的编程是很接近的。

#### 2. 视频适配器

为了显示文本或图形，计算机使用了所谓的视频适配器 (Video adapter)，也即一块连接计算机和显示器的电路板。由于计算机可以连接多种不同的显示器，而每种显示器的特性以及编程方式都很不相同，因此也就存在若干种不同的视频适配器。大部分计算机上有一个单色适配器或者彩色图形适配器 (CGA)，现在增强图形适配器 (EGA) 和 VGA 也越来越常见了。在本章中我们把讨论的焦点集中于单色适配器和彩色图形适配器，但所讨论的概念对于增强图形适配器也是适用的。

单色适配器只和 IBM 单色显示器一同工作，单显不能产生彩色，也不能产生图形信息，换句话说，它只能在文本方式下工作，只能显示文本字符数据以及描画由字符构成的简单图形。

彩色图形适配器可以和许多不同的显示器一同工作，例如它可以与 RGB (红绿兰) 监视器之类的专用显示器连接，也可以与合成彩色显示器连接，甚至可以与黑白监视器连接，当然这时候黑白监视器是与彩色信号连接，但不能利用全部彩色信息，因此不仅损失彩色能力而

而且还损失了 IBM 单色显示器下高质量的清晰度。

值得注意的是,黑白监视器虽然可以认为是单色的,但不能与单色适配器的起使用。在 IBM PC 词汇中,“单色”仅指 IBM 单色显示器。所以专门讨论单色适配器的内容只适用于单色显示器,不适用于任何彩色选件,也不适用于在彩色图形适配器下使用黑白监视器的情形。现样,凡专门讨论彩色图形适配器的内容都不适用于单色显示器,但适用于所有彩色显示器,除了彩色信息本身,绝大多数内容也适用于黑白监视器。

彩色图形适配器既可以在文本方式下工作,也可以在图形方式下工作。

### 3. 字符与象素

在文本方式下,计算机的最小可控制显示元素是字符,视频显示器一满屏可以显示 25 行每行 80 个字符(列),一共 2000 个字符。只要给出适当的字符编码,字符的描画则完全是由计算机自动进行的。

在图形方式下,计算机的最小可控制显示元素是象素。象素也是计算机可显示的最小图形元素,屏幕上的一个字符就是由许多象素按字符形状排列构成的。每个象素都可以亮或者不亮,象素的大小和位置都经过仔细的调整,因此象素之间没有间隔,并且充满整个屏幕,如果一部分相邻的象素都是亮的,它们就不会呈现为一些离散的点而是一个实心的明亮区域。如果象素点足够小,象素图形显示就非常清晰和精确。不过在大多数象素显示屏上,这些象素点处于粗糙与细微之间,即足以形成一个象样的图像,但有时看增来又有些原始。

### 4. 前景与背景

前景和背景是控制显示彩色时用到的概念。无论在文本方式下还是在图形方式下,都要区分前景颜色和背景颜色。所谓前景颜色是指所显示的字符或图形(角素)本身的颜色,而背景颜色则是指围绕所显字符或图形的四周空间的颜色。

在文本方式下,配有彩色图形适配器的彩色显示器最多可显示 16 种不同的颜色,如表 10-1 指示;而单色显示器则用不同的可见属性(高亮度、下划线或反视频等)来代替彩色。

注意,只有前八种颜色可用于背景,而所有十六种颜色均可用于前景(即字符本身)。

颜色标识	前景或背景
BLACK(黑)	均可
BLUE(兰)	均可
GREEN(绿)	均可
CYAN(青)	均可
RED(红)	均可
MAGENTA(洋红)	均可
BROWN(棕)	均可
LIGHTGRAY(浅灰)	均可
DARRKGRAY(深灰)	只限前景
LIGHT BLUE(淡兰)	只限前景

LIGHT GREEN(浅绿)	只限前景
LIGHT CYAN(浅青)	只限前景
LIGHTRED(淡红)	只限前景
LIGHTMAGENTA(浅洋红)	只限前景
YELLOW(黄)	只限前景
SHITE(白)	只限前景

表 10-1 文本方式下可用颜色表

在图形方式下,象素也可以配上颜色(前景),同时也可以配置象素的背景颜色。但象素的前景和背景颜色设定都需通过所谓的调色板(Palette)进行。调色板是一个颜色表,它是硬件能显示的颜色子集,这个子集中的颜色是能够同时显示的。一般彩色视频适配器都有一组调色板供编程者选择需要同时显示的不同颜色集合。而背景颜色总是对应调色板中的第 0 项。

由于图形硬件的差别,图形方式下的颜色控制对于不同适配器是相当不同的。在以后的讨论中还会更详细地涉及这些问题。

#### 5. 坐标

无论在文本方式还是图形方式,计算机都通过某种坐标系来指定屏幕显示器上的特定位置。在文本方式下,屏幕被分成 25 行 X 80 列个显示位置,因此坐标就定义为 X 和 Y 二维参照系。根据约定,X 表示列位置而 Y 表示行位置,(以 x:y 的形式给出),X 的值从左向右增加,Y 的值从上到下增加。所以监视器的左上角的坐标是 1:1,侧右下角的坐标是 80:25。为了指出某一时刻所处理的屏幕位置,系统还引入了光标的概念。通常为一个亮条或小亮块。

在图形方式下,显示器的象素也是按水平和垂直排列成行的。在不同计算机的图形方式下的主要区别是象素的大小不同。因此象素也通过 x-y 坐标系来选定,但对不同分辨率的显示方式,坐标的取值有所不同。在 CGA 的低分辨率方式,象素相当大,水平方向只能容纳 320 点,垂直方向为 200 点,因此其坐标系如下图所示:

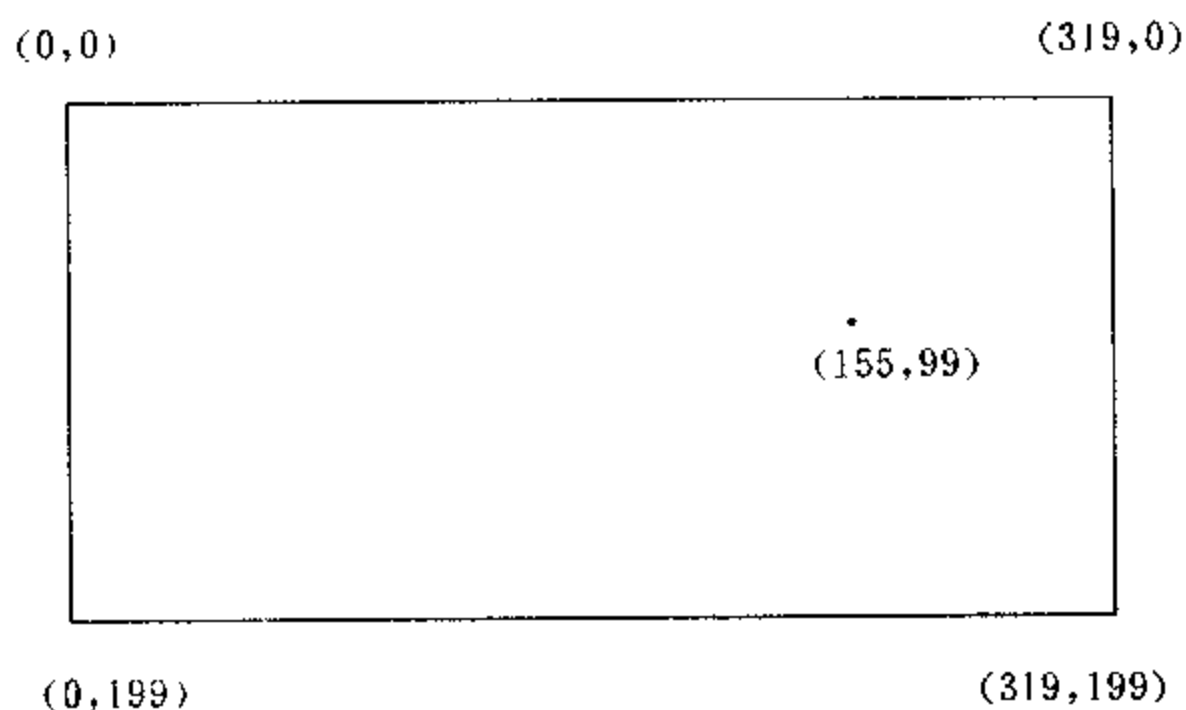


图 10-1 CGA 的坐标系

对于相对新型的视频显示控制卡 VGA 适配器来说,就可以具有较高的分辨率。象素也更小,水平方向具有 640 点而垂直方向有 480 点,从而坐标系统也会有些差异。象素越小,每帧映象所含象素也就越多,图形显示的质量就相应地更高。

现在我们可以明白什么图形方式下工作时了解所安装的适配器以及当前的工作方式是多么重要!由于屏幕的坐标系统在不同适配器之间、在同一适配器的不同方式之间都是不一样的,如果你想在屏幕右下角描画某种图案,就必须知道该处的坐标是多少!

许多图形系统支持当前指针(CP)的概念,CP 除了不可见之外与文本方式的光标非常类似,也指出当前的显示处理位置。

## 6. 窗口与视口

窗口(Window)是文本方式下的一个概念。通常的程序设计中,你会打算利用 PC 机的整个  $80 \times 25$  的视频显示区。可是在有些时候可能需要把输出限制在屏幕的一定区域内。窗口就是在文本方式下在 PC 的视频屏幕上定义的一个矩形区域,当程序往屏幕上写东西时,它的输出被限制在这个活动的窗口中,屏幕上剩下的区域则是无法直接访问的。

在图形方式下,在 PC 的视频屏幕上也可以定义一个矩形区域,这就是视口(Viewport)。当程序输出图形时,视口就是实际的屏幕,屏幕视口之外的剩余部分同样是无法直接访问的。

## 7. 显示内存区

我们已经知道 PC 机的显示器可以显示文本或者图形,在显示中还可以指定前景和背景颜色等等显示属性。计算机用于存贮显示的字符(或图形)以及颜色信息的那部分特殊的内存区域就叫做显示内存,有时称为视频缓冲区。通过它可以告诉计算机显示什么以及以何种颜色显示。虽然显示内存处于视频适配器上,但仍被认为是 RAM 的一部分。以文本方式为例,显示内存的第一个字节包含监视器上的第一个字符,显示在左上角。这样,如果显示内存第一个字节包含数值 41h(字母“A”的十六进制 ASCII 码值),那么监视器就会在左上角显示字母“A”。显示内存的第二个字节是第一个字符的属性字节,包含色彩和其它显示信息。

在文本方式下这种“字符、属性;字符、属性”的模式将重复 2000 次,因为监视器上可以显示 2000 个字符。换句话说,一个显示屏的内容需要 4000 字节的视频缓冲区。

属性字节有四种显示控制特性:背景颜色、前景颜色,前景亮度和闪烁。前面已经介绍,文本方式下的彩色图形适配能够控制显示 16 种不同的颜色,而每一种颜色实际上是包含了四种元素:蓝、红、绿和亮度。计算机显示的颜色取决于这些元素的不同组合。例如黑色不包含任何元素,浅青色则使用了蓝、绿和亮度三种元素。视频适配器还支持闪烁特性,它使字符明灭交替闪亮,但它不影响颜色。

属性字节的格式如图 10-2 所示。

属性字节的前三位(0,1,2)控制一个字符的前景颜色,而第 4 位(bit3)控制(前景)颜色的亮度,因此你可以用这四位的组合建立起 16 种不同的前景颜色。位 4,位 5,位 6 决定了字符的背景颜色,而位 7 若置位则使字符闪烁。由于背景无亮度元素,因此只有八种可用的颜色。

在单色显示器上,属性字节只能建立五种显示特征:隐藏、正常、高亮度,下划线以及反视频显示。

当前景与背景颜色相同时,字符就看不见了。亮度与闪烁位对隐藏没有影响。

反视频显示是在亮色背景下显示深色字符,故可以将背景设置成浅灰(不加亮度的白色)

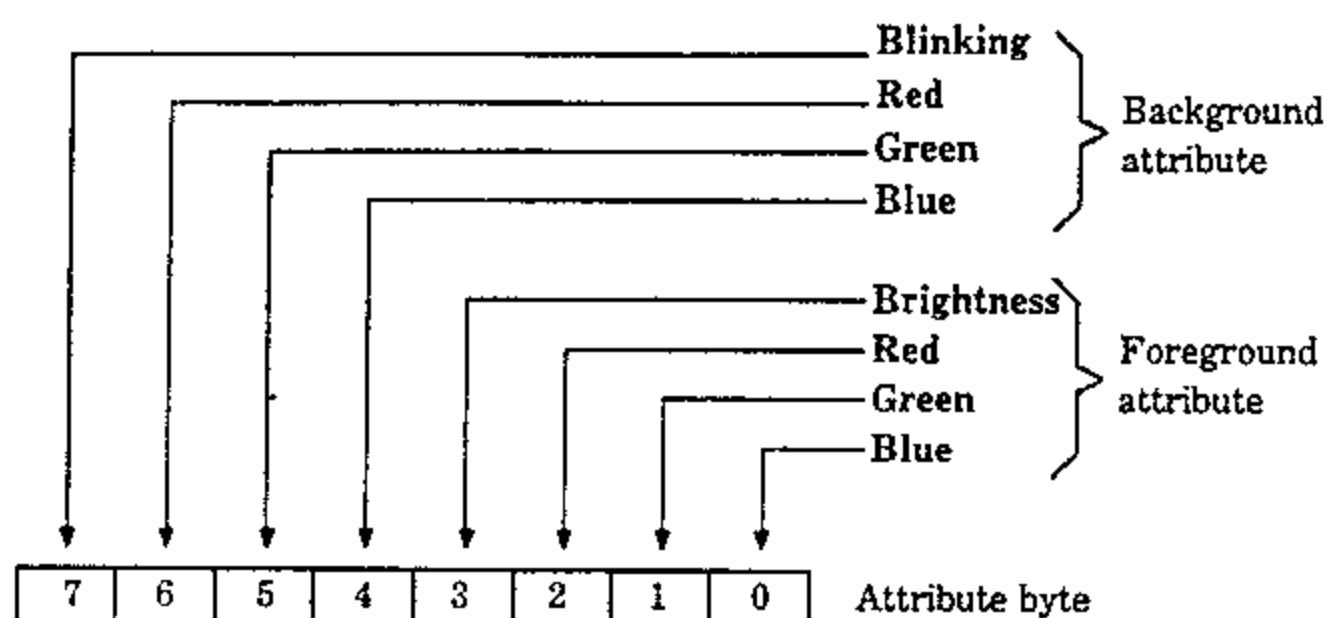


图 10-2 视频属性字节的格式

以及将前景置为黑色。反显字符无法加亮,但可以闪烁。

下划线是为单显适配器专门设置的,对应的颜色设置是蓝色前景、黑色背景。带下划线的字符能够加亮或闪烁,但无法反显。

图形方式的属性控制和显示内容指定不同于文本方式,这里就不再详细介绍了。

有了前面介绍的有关视频显示的概念,进一步讨论使用 Turbo Pascal 进行视频编程就不会难以理解了。

## § 10.2 使用 Turbo Pascal 显示文本

从本节开始我们讨论如何用 Turbo Pascal 来控制计算机视频显示器以充分发挥其潜在的能力。首先讨论比较容易但非常有用的视频特性:文本显示。

### § 10.2.1 方式、颜色和位置控制

个人计算机支持 16 种不同的显示方式,其中有五种用于显示文本,它们列于表 10-2 中。这些方式控制着显示器上字符的尺寸和颜色,例如,方式 0 是在黑白显示器上显示大字符(每行 40 个字符)。

方式	大小	色彩	适配器
0	40×25	黑白	CGA, EGA
1	40×25	彩色	CGA, EGA
2	80×25	黑白	CGA, EGA
3	80×25	彩色	CGA, EGA
7	80×25	黑白	单色适配器

表 10-2 个人计算机的文本方式

前 4 种显示方式, 0 到 3, 只能在彩色图形适配器和增强彩色图形适配器上工作, 大多数这类适配器都能在这 4 种方式之间切换。但方式 7 仅由单色适配器使用。

我们可以用 Turbo Pascal 的 TextMode 命令来改变计算机的文本方式, 这条命令仅对彩色图形适配器有效, 对单色适配器不起作用。该命令可以带有参数也可以不带参数, 如果不带参数, textMode 将把视频显示中设置成先前使用的文本方式; 如果带有参数, 则 TextMode 依据参数将视频显示器设置成 PC 所支持五种文本方式的前四种方式之一。Turbo Pascal 为 TextMode 命令提供了四个标准整型常数: BW40, BW80, CO40 和 CO80。有一点要特别注意, TextMode 在改变方式之前总是先清除当前的屏幕。TextMode 命令的设置结果参见表 10-3

允许你使用 TextColor 和 TextBackground 命令为文本选择前景和背景颜色。彩色图形适配器提供了 16 种供选取的颜色, 并为每种颜色提供了一个色彩标识条, 参见表 10-4。TextColor 用于设置前景(字符的)颜色, 它能使用全部 16 种颜色, 而 TextBackground 命令用于设置背景颜色, 它只能使用深颜色。

参数	设置结果
TextMode	设置成先前的文本方式
TextMode(BW40)	黑白/每行 40 字符
TextMode(BW80)	黑白/每行 80 字符
TextMode(CO40)	彩色/每行 40 字符
TextMode(CO80)	彩色/每行 80 字符
Turbo Pascal 标准常数值	
BW40=0	
BW80=1	
CO40=2	
CO80=3	

表 10-3 Turbo Pascal 的 TextMode 命令

深颜色	浅颜色
0: Black (黑)	8: DarkGray (深灰)
1: Blue (兰)	9: LightBlue (浅兰)
2: Green (绿)	10: LightGreen (浅绿)
3: Cyan(青)	11: LightCyan (浅青)
4: Red(红)	12: LightRed (浅红)
5: Magenta (洋红)	13: LightMagenta(浅洋红)
6: Brown (棕)	14: Yellow (黄)

表 10-3 Turbo Pascal 色彩标识符

Turbo Pascal 提供了称为 Blink 的标准常数, 将它加到前景颜色上时, 则会引起字符闪烁。这里有几个关于如何使用 Turbo Pascal 的颜色命令的例子:

```
textColor(Cyan);
textBackground(Blue);
TextColor(Cyan+Blink);
```

一旦调用了上述命令, 新的颜色将在你显示下一字符时起作用, 已经显示在屏幕上的字符则不受影响, 即保持其颜色不变。

本节介绍的最后一条命令是关于如何控制文本在屏幕上的显示位置的定位命令。

我们知道可以通过屏幕坐标系统在屏幕上定位, 而且还有一个光标指示着当前的位置选择。Turbo pascal 允许你使用 GotoXY 命令把光标移到屏幕上的任何光标位置。例如, 命令 GotoXY(1,10) 将把光标定位在第 10 行的第 1 列。如果你想知道光标光前所处位置的坐标, 可以使用 Turbo Pascal 的 WhereX 和 WhereY 函数。whereX 返回一个整数指出光标所在的列, 而 WhereY 则返回光标所在的行。

Turbo Pascal 的标准屏幕命令使你能够控制计算机去建立有吸引力而且富含信息的屏幕显示下面的程序说明如何用所有至今为止描述过的命令去建立一个简单的数据输入例程。我们已经知道的命令有: textMode, TextColor, TextBackground, GotoXY, WhereX, 和 WhereY。

```
Program Box;
Uses CRT;
Var
  i, code,
    x, y : Integer;
    s : String[20];
  ( * * * * * )
```

```
Procedure DrawBox(x1,y1,x2,y2,fg,bg : Integer);Var
  i : byte;
```

```
  Begin
    TextColor(fg);
    textBackground(bg);
    For i := (x1+1) To (x2-1) Do
      Begin
        GotoXY(i,y1);
```



```

Write( # 205);
GotoXY(1,y2);
Write( # 205);
End;

```

```

For i := (y1+1) To (y2-1) Do

```

```

  Begin
    GotoXY(x1,i);

    Write( # 186);
    GotoXY(x2,i);
    Write( # 186);
    End;

```

```

GotoXY(x1,y);
Write( # 201);
GotoXY(x2,y1);
Write( # 187);
GotoXY(x1,2y);
Write( # 200);
GotoXY(x2,y2);
Write( # 188);
End;

```

```

( * * * * *
* )

```

```

Procedure GetNumber;
Begin
  DrawBox(5,14,16,White,Black);
  GotoXY(7,15);
  Write( 'Enter a number (1-10): ');
  x := WhereX;
  y := WhereY;
  Repeat
    GotoXY(x,y);
    s := '
  Write(s);
  GotoXY(x,y);
  ReadLn(s);

```

```

        Val(s,i,code);
        Until ((i > )) And (i < 11))) And (code = 0);

End;

( * * * * * )

Begin
  ClrScr;
  textMode(c40);
  GetNumber;

  ClrScr;
  TextMode(c80);
  GetNumber;
  GotoXY(1,20);
  WriteLn;
  Write( 'Press ENTER... ');
  ReadLn;
End.

```

过程 DrawBox 用标准的 ASCII 图形字符在屏幕上的任意位置画出一个矩形。该过程定义了屏幕上的一个数据输入区域。

## § 10.2.2 直接存取视频存储区

把显示信息送到显示屏上有两种基本方法。一种是标准 Turbo Pascal 命令 Write 使用的方法，它利用 BIOS 中断将字符发送到监视器。另一种方法，通常是更有效率的方法是，绕过 BIOS 直接地将字符写入显示存储区。这种方法比第一种方法要快得多，但由于垂直回扫在 CGA 适配器上可能会产生“雪花”，也就带来了某种复杂性。如果你要你的程序直接写视频存储区，你就必须采取一些特别的步骤以消除雪花。可庆幸的是，Turbo Pascal 用 CRT 单元的形式提供了一个简便的解决方案。如果你在程序中使用了 CRT 单元，Turbo pascal 就会自动地控制住视频存储区直接写入而不产生雪花。

虽然 Turbo pascal 的 CRT 单元极其有用，但它的视频能力也有某些限制，例如，即使在直接向视频存储区写入的时候，CRT 单元似比你得到的最高速度要慢。本节将提供几个汇编例程，使你的程序能得到那种极高的速度。另外，Write 命令总是将光标移到屏幕的下一位置，因此，如果你向屏幕右下角写一个字符，就会引起显示器向上滚动一行。这就使得用 Turbo pascal 语句写出整个屏幕成为不可能。本节的后面部分将揭示我们的计算机视频显示器的技术细节，并给出如何才能在程序中驾驭它的技巧。

在使用显示存储区之前，我们必须知道它在哪儿。对于单色适配器。显示存储区起始于 B000H 段，而对于彩色图形适配器，它起始于 B800h 段。由于各适配器使用视频存储器的不同区域。因此我们必须弄清所用显示适配器的类型。判定所用适配器的一种方法是通过一个

BIOS 中断，这种技术将在下一章(第十一章)用 CurrentVidMode 过程来说明。另一种得到同样信息的方法如下所示：

```
Function VidSeg ; Word;
Begin
  If Mem[ $ 0000; $ 0449]=7 Then
    VidSeg := $ B000
  Else
    VidSeg := $ B800;
End;
```

Vidseg 函数检查存储器 0000h 段中偏移量为 0449h 位置上的单元，这里 DOS 存储着视频适配器代码。如果该位置的字节值为 7，适配器就是单色的，如果不等于 7，那么就可以安全地假定适配器是彩色图形的。

虽然这种方法比用 BIOS 中断更直接，但它对 IBM-PC 的兼容性高度敏感：如果用户的计算机将这个信息存放在不同的存储器地址，这种方法就失灵了。很幸运，大多数 PC 制造商都仔细地将存放视频显示器类型的位置保持与 IBM PC 存放该信息的位置相同。

一旦我们知道了所用视频适配器的类型，绕过 BIOS 直接将字符写入显示存储区就很容易了。下面的程序可以判定所用的适配器类型，然后将字母“A”用黑背景白字母的形式充填显示存储区：

```
{ $ R- , S- }
Program FillScreen;
Uses CRT;
Var
  CharAtt
  I, j;
  vs; Word;
( * * * * * )
Function VidAeg; Word;
Begin
  If Mem( $ 0000; $ 0449)=7 Then
    VidSeg := $ B800;
  Else
    VidSeg := $ B000;
  End;
( * * * * * )

Begin
  ClrScr;
  VS := VidSeg;
  j := 0;
  ( * $ 41 is the character 'A' * )
```

```

(* $70 is White on Black *)
CharAtt:=( $70 Shl 4)+ $41;

For i:=1 To 2000 Do
  Begin
    MemW[VS:j]:=CharAtt; (* Write character and attribute *)
    Inc(j,2);             (* Point to next video character *)
  end;
GotoXY(1,1);
Write('press ENTER... ');
ReadLn;
End.

```

程序中使用 VidSeg 函数将视频显示存储区的段号存入整型变量 VS 中，For-Do 循环使用标准数组 MemW 将字符和属性字节传送到视频存储区，由于 MemW 作用于一个整字(16 位)，因此属性字节和字符必须组合到单个字中。字符设置成 41h，这是字母“A”的 ASCII 代码，属性字节设置成 70h，它在黑色背景上产生一个白字母。变量 j 保持着在显示存储区中的偏移量，它在每个字符和属性写过之后递增 2。

用下列的过程 FastWrite，你能直接在指定坐标 x 和 y 处将字符串连同颜色属性一起写入显示存储区。

```

Procedure FastWrite(x,y:Byte; VAr S:String; Fg,bg:Byte);
Var
  W: Word;
  i, ColAtt:Byte;
Begin
  ColAtt:=(bgShl 4) +fg; (* 建立属性字节 *)
  W:=((y-1)* 80+(x-1))* 2; (* 计算偏移量 *)
  For i:=1 To Length(s) Do
    Begin
      MemW[Vs:w]:=(ColAtt Shl 8) +Ord (s[i]);
      Inc(W,2);
    end;
  end;
End.

```

过程一开始先建立属性字节(ColAtt)，将背景颜色左移 4 位再加上前景颜色。接着，过程用以下公式计算在显示存储区中的起始偏移量：

$$X:=((y-1)* 80+(x-1))* 2;$$

在循环的每次迭代中，Turbo Pascal 的 MemW 数组同时设置字符字节和属性字节，过程通过将颜色字节左移 8 位再加上字符字节将它们组合起来。注意这将把属性字节置于字的高位部分。而在显示存储区中属性字节是放在字符字节之后的。这里要再一次提醒大家：Inted 的反向存储法迫使我们在按字操作时要逆向思考。

下面我们来看一看为什么要避免垂直回扫。如果你使用的是 CGA 视频适配器，那么试着

运行一下以下的程序，它用 Turbo pascal 的 FastWrite 过程在监视器上显示大“V”。你立即会发现避免垂直回扫是非常重要的。

```

{ $R- }
Program FastWriteDemo;
Uses CRT;
Var
    Vs : Word;
    i, j : Byte;
    s : String;
    ( * * * * * )
Function VidSeg : Word;
Begin
    If Mem[ $ 0000 : $ 0449 ] = 7 Then
        VidSeg := $ B000
    Else
        VidSeg := $ b800;
End;
    ( * * * * * )
Procedure FastWrite(x, y : Byte;
                    Var s : String;
                    fg, bg : Byte);
Var
    w : Word;
    i, ColAtr : Byte;
Begin
    ColAtr := (bg shl 4) + fg;    ( * create attribute byte * );
    w := ((y - 1) * 80 + (x - 1)) * 2;    ( * calculate offset * )
    For i := 1 To Length(s) Do
        Begin
            MemW[VS : w] := (ColAtr Shl 8) + Ord(s[i]);
            Inc(w, 2);
        End;
    End;
    ( * * * * * )
Begin
    ClrScr;
    VS := VidSeg;
    s := 'XXXXXXXX';
    j := 5;

```

```

For i:=1 To 25 Do
  Begin
    FastWrite(j,i,s,Yellow,Black);
    Inc(j,i);
  End;
GotoXY(1,24);
Write('Press ENTER...');
ReadLn;
End.

```

当我们用 CGA 适配器运行上述程序时，我们会注意到屏幕上有一些“雪花”。这是由垂直回扫行起的。垂直回扫是每秒钟进行许多次的由显示内存更新屏幕的过程，每次垂直回扫大约需要 1.25 毫秒。如果我们直接向显示内存传送数据而垂直回扫正在进行，就会出现雪花。

避免垂直回扫的唯一办法是在两次垂直回扫之间写显求内存，换句话说，程序必须等待回扫结束再向显示存储区写字符，并且在下一次回扫开始前写毕。要做到这一点用 Turbo Pascal 就不够快了。我们必须使用外部汇编过程或嵌入码 (inline code)。

下面所列的是一个汇编过程，它可以不产生雪花地向视频内存写出一个字符串。这个汇编程序使用了 Turbo Pascal 的专门特性，因此必须使用 TASM·EXE 来汇编它。

```

• MODEL TPASCAL
• DATA
VIDTYPE DB?
• CODE
FastWrite PROC FAR x: BYTE, y: BYTE, S:DWORD,
fg: BYTE, bg:BYTE
  PUBLIC FASTWRITE
FASSTR:
;.....
; 计算进入显示内存的偏移量((y-1)*80+(x-1))*2
;.....
  PUSH DS

XOR     BX, BX
XOR     AX, AX
MOV     BL, X           ; 从堆栈中得到 X;
MOV     AL, y           ; 从堆栈中得到 y;
DEC     BX              ; 递减 x 和 y 以得到正确
DEC     AX              ; 的偏移量
MOV     CX, 0080        ; y(在 AX 中)乘以 80(在 CX 中)

```

MUL	CX	
ADD	AX, BX	; x 加到 y 上;
MOV	CX, 0002	; 将(x+y 的)和乘以 2。
MUL	CX	
MOV	DI, AX	; 将起始偏移量存在 DI 中
;.....		
;建立属性字节		
;.....		
MOV	BL, bg	;得到背景颜色
MOV	AL, fg	;得到前景颜色
MOV	CL, 4	;将前景颜色移进
SHI	BX, CL	; 高半字节
ADD	BX, AX	;加进前景
XCHG	BL, BH	;送到高字节
MOV	DX, 3DAH	; 把 CRT 口地址装入;DX 中。
;.....		
;得到监视器的类型		
;.....		
XOR	AX, AX	; 将 000h 赋给 ES
MOV	ES, AX	
MOV	AX, 0449h	;赋予视频类型单元
MOV	SI, AX	;的偏移量
MOV	AX, ES:[SI]	; 如果视频类型是 7
CMP	AL, 7	;那么监视器是单色的。
JZ	MONO	
MOV	AX, 0B800H	;将彩色段装入 ES
MOV	ES, AX	
MOV	VIDTYPE, 1	
JMP	CHKSTR	; 继续

```

MONO:  MOV AX, 0B000H    ;将单色段装入
MOV     ES, AX           ;ES 中。
MOV
VIDTYPE, 0

;.....
;装入字符串并检查串长度是否为零
;.....

CHKSTR:  LDS     SI, S      ;把字符串的地址装入 DS:SI
MOV      CL, [SI]         ;串长度送入 CL
CMP      CL, 0            ;如果串长度为 0,
JZ       ENDSTR           ;退出本过程。
CLD                          ;清除方向标记

NEXTCHAR: INC     SI       ;指向下一字符,
MOV      BL, [SI]         ;将它送入 BL,
CMP      VIDTYPE, 0       ;如果是单显, 则不检
JE       MOVECHAR         ;查回扫。

WLOW:    IN      AL, DX    ;得到 CRT 状态
TEST     AL, 1            ;测试回扫状态
JNZ      WLOW             ;如果不在回扫则等它开始
CLI                          ;关闭中断

WHIGH:   IN      AL, DX    ;得到 CRT 状态
TEST     AL, 1            ;测试回扫状态
JZ       WHIGH            ;如在回扫则等它停止。

MOVECHAR: MOV     AX, BX   ;将颜色和字符送入 AX;
STOSW                      ;将颜色和字符送向屏幕。
STI                          ;现在允许中断发生

LOOP     NEXT CHAR        ;做完了吗?

ENDSTR:  POP     DS
RET

FASTWRITE ENDP

```



```
CODE      ENDS

          END
```

Fast Write 过程检查 3DAh 号适配器 I/O 口，以得到称为回扫同步信号的状态位，当此状态位置位时，垂直回扫正在进行。过程首先查看监视器是否处于两次回扫之间（状态位等于 0），如果是，循环等待回扫开始，然后过程再等待回扫结束。

一旦回扫结束，过程就将字符传送到显示内存中。因为过程恰好等待到回扫结束就开始传送字符，因此在下一次回扫开始之前应该有足够的时间用于这种传送。

用 Turbo Assembler 的 TASM 汇编这段代码，使之成为目标文件 (FASTWRIT.OBJ) 并将其说明成外部例程，象下面的程序清单中给出的这样：

```
{ $R- ,F+ }
Program FastWriteDemo;
Uses CRT;
Var
    Vs : Word;
    i,j : Byte;
    s : String;
{ $L FASTWRIT }
Procedure FastWrite(x,y:Byte;
                    Var s:String;
                    fg,bg:Byte); External;

Begin
    CtrSer;
    VS := VidSeg;
    s := 'XXXXXXXX';
    j := 5;

    For i := 1 To 25 Do
        Begin
            FastWrite(j,i,s,Yellow,Black);
            Inc(j,i);
        End;

    For i := 1 To 25 DownTo 1 Do
        Begin
            FastWrite(j,i,s,Yellow,Black);
            Inc(j,1);
        End;
    GotoXY(1,24);
```

```
Write( 'Press ENTER...' );
Readln;
End.
```

在 FASTWrite 中所用的技术在大多数 IBM 兼容个人计算机上都能工作, 它大大改进了视频显示的速度。

### § 10.2.3 Turbo Pascal 的窗口程序设计

一般情况下, 我们的程序将希望利用 PC 机的整个  $80 \times 25$  的视频显示屏幕, 但是偶尔有些时候, 需要限制输出只显示于监视器上一部分的时候, 我们可以使用 Turbo Pascal 的 Window 命令。例如:

Window(10,10,20,15)限制我们的程序的显示处于一个起点在第 10 行第 10 列的  $10 \times 5$  的矩形区域内。显示器的余下部分对 Write 命令来说是关闭的。(注意 FastWrite 过程不理睬 Turbo Pascal 且能写到活动窗口之外。)当我们想回到整屏窗口时, 命令

```
Window(1,1,80,25)
```

将使监视器回到正常操作状态。

Window 命令的最有用的特性是它的重新安排坐标系统以适合活动窗口的能力。也就是说, 屏幕坐标仅参照活动窗口而不是整个屏幕。因此命令

```
GotoXY(1,1)
```

将光标定位在活动窗口的左上角, 而不是整个屏幕的左上角。事实上, Turbo Pascal 看待窗口好象它就是整个屏幕: 当文本移出窗口底部时, 屏幕上滚一行。

#### § 10.2.3.1 弹出窗口

虽然 TurboPascal 的 Window 命令是很有用的, 但它还是太简单了, 以致缺乏足够的能力去建立看上去有专业风格的弹出窗口。问题在于 Turbo Pascal 窗口擦去它所用的屏幕区域的内容。大多数人希望窗口能象在流程序 Side Kick 中那样工作, 也就是说, 当窗口消失时, 屏幕上窗口“后面”的原先的文本能够重新显示。

为了建立真正的弹出式窗口, 需要在我们打开窗口之前保存屏幕上的内容, 然后在我们关闭弹出窗口时恢复屏幕。

为了保存屏幕, 需要定义一个变量, 它能存贮在屏幕上的所有信息: 2000 个字符, 2000 个颜色属性, 以及光标的 x, y 坐标。在下列数据结构 Screen Type 中提供了我们的全部需要。

```
Type
Screen Type=Record
Pos:Array[1..80,1..25] of Record
Ch : Char;
At : Byte;
End;
CursX,
cursY:Byte;
End;
```

Screentype 是一个嵌套 Record 的数据类型, 它在名为 Pos 的数组里存放字符和属性, 数组的维数与监视器的坐标匹配, 因此, 引用在 20 行 10 列的字符就可以用下列语句:

```
Screen.Pos[10,20].Ch
```

整型变量 CursX 和 cursY 用于存放光标位置。

ScreenType 变量相当有用，我们不仅可以用它存贮屏幕的内容，而且也可以在内存中修改该变量的内容，然后再直接把它传送到视频显示，瞬间更新整个视频显示。

使用 screenType 变量存贮和恢复视频图象有几种方法。一种是把变量说明成 Absolute 处于显示内存位置。如果你有一个彩色图形适配器，你就这个说明：

```
Var
```

```
Screen: Screen Type Absolute $B800;
```

不幸的是，这种方法需要我们进一步选择偏移量，这意味着只能服务于一种视频适配器类型(除非我们为单显和彩显分别定义屏幕)。一种更好的方法是把屏幕定义成一个 Pointer 变量，然后程序再把指针置成显示内存中的正确的偏移量，这可以取决于所用的适配器。下面的程序说明了这种技术：

```
Program Window Pointer;
```

```
Uses CRT;
```

```
Const
```

```
    MaxWin = 5;
```

```
Type
```

```
ScreenPtr = ^ScreenType;
```

```
ScreenType = Record
```

```
Pos: Array[1..80, 1..25] of Record
```

```
    Ch: Char;
```

```
    At: Byte;
```

```
End;
```

```
End
```

```
Var
```

```
    Screen: ScreenPtr;
```

```
( * * * * * )
```

```
Function VidSeg: Word;
```

```
Begin
```

```
If Mem($0000, $0449) = 7 Then
```

```
    VidSeg := $B800;
```

```
Else
```

```
    VidSeg := $B800;
```

```
End;
```

```
( * * * * * )
```

```
Begin
```

```
ClrScr;
```

```
Screen:=Ptr(VidSeg,$0000);
Screen.Pos[1,1].ch:='A';
```

```
ReadLn;
```

```
End.
```

程序从清除屏幕开始,然后用Ptr命令让Screen变量指向显示内存的正确位置,由于这一处理,任何对Screen变量的修改都将显示在我们的监视器上。

用这种方法时,不要对Screen变量使用Dispose命令,否则TurboPascal就会尝试从显示适配器中释放内存供重新分配,很有可能使我们的程序崩溃。

### §10.2.3.2 多逻辑屏幕和弹出窗口

只要程序内存中放得下,我们的程序就可以有许多个屏幕变量。保存在内存中不显示的屏幕常常称为逻辑屏幕以区别于(计算机监视器的)物理屏幕。一个程序可以写到逻辑屏幕而丝毫不干扰物理屏幕的显示。然后,在我们需要显示逻辑屏幕时,简单地把它的内容送到物理屏幕上。

逻辑屏幕的用法最好用例子来解释。下面这个程序用了一个物理屏幕变量(Screen)以及三个逻辑屏幕变量(Screen1,Screen2和Screen3)。通过在键盘上打入1,2或3,我们就能显示三个逻辑屏幕之一。

```
Program FastScreenDemo;
```

```
Uses CRT;
```

```
Const
```

```
    MaxWin=5;
```

```
Type
```

```
    ScreenPtr=^ScreenType;
```

```
    ScreenType=Record
```

```
        Pos:Array[1..80,1..25] of Record
```

```
            ch:Char;
```

```
            At:Byte;
```

```
        End;
```

```
Var
```

```
    Screen,
```

```
    Screen1,
```

```
    Screen2,
```

```
    Screen3;screenPtr;
```

```
    Ch:Char;
```

```
    i,j:Byte;
```

```
(*****)
```

```
Function VidSeg:Word;
```

```
Begin
```

```

If Mem($ 0000; $ 0449) = 7 Then
    VidSeg: $ B000
Else
    VidSeg := $ B800;
End;
( * * * * * )
Begin
    New(Screen1);
    New(Screen2);
    New(Screen3);

    For i := 1 To 80 Do
        For j := 1 To 25 Do
            Begin
                Screen1^.Pos[i,j].Ch := '1';
                Screen2^.Pos[i,j].Ch := '2';
                Screen3^.Pos[i,j].Ch := '3';
                Screen1^.Pos[i,j].Ch := '$ 07';
                Screen2^.Pos[i,j].Ch := '$ 07';
                Screen3^.Pos[i,j].Ch := '$ 071';
            End;
        ClrScr;
        Screen := Ptr(VidSeg, $ 0000);

        Repeat
            GotoXY(1,1);
            Write('Press 1,2,3 to change screens or 0 to exit. ');

            Ch := ReadKey;
            Case Ch Of
                '1': Screen := Screen1;
                '2': Screen := Screen2;
                '3': Screen := Screen3;
            End;
        Until Ch = '0';
    End.

```

叠加到显示内存的 Screen，作用如同物理设备。程序通过将物理设备变量 Screen 设成等于逻辑屏幕之一来改变显示内存。转换是迅速的，整个屏幕在一条语句中被更新。程序也能用一条语句将物理屏幕的内容传送到逻辑屏幕上，象这样：

```
Physical Screen := LogicalScreen;
```

操纵逻辑屏幕是建立弹出窗口所需要的技术。在我们打开一个弹出窗口之前，要将物理屏幕的付本保存在一个逻辑屏幕变量中。然后，在我们关闭这个窗口时，要把屏幕恢复成原先的样子。这样一来，我们的窗口就可以从任何地方弹出，而且当不再需要时可以不留痕迹地消失。

下面的程序使用一个逻辑屏幕数组建立多达五个弹出式窗口。当我们运行这个程序时，将会看到窗口能不引起任何问题地覆盖。

```
( * $ R - * )
Program WindowDemo;
Uses CRT;
Const
    MaxWin=5;
Type
    ScreenType=Record
        Pos:Array[1..80,1..25] of record
            Ch:Char;
            At :Byte;
        End;
        CursX;
        CursY:Byte;
    End;

WindowPtr=^ WindowType;
WindowPtr=Record
    Scr:ScreenType;
    WinX1,
    WinX1,
    WinX2,
    WinY2 :Byte;
End;

Var
    Ch:Char;
    I:Integer;
    ActiveWin:WindowPtr;
    Windo:Array[0..MaxWin] of WindowPtr;
    CurrentWindow :Integer;

( * * * * * )
Function VidSeg:Word;
Begin
    If Mem($ 0000:$ 0449)=7 Then
```

```

    VidSeg: $B000
Else
    VidSeg := $B800;
End;
( * * * * * )
{SL FASTWRIT}
Procedure FastWrite(x,y:Byte;
                    s:String;
                    bc,fc:Byte);External;
( * * * * * )
Procedure FastBox(x1,y1,x2,y2,fg,bg:Byte);
Var
    i:Byte;
    s:String[i];
Begin
    TextColor(fg);
    TextBackground(bg);

    s := #205;
    For i := (x1+1) to (x2-1) Do
        Begin
            FastWrite(i,y1,s,fg,bg);
            FastWrite(i,y2,s,fg,bg);
        End;

    s := #186;
    For i := (y1+1) to (y2-1) Do
        Begin
            FastWrite(i,x1,s,fg,bg);
            FastWrite(i,x2,s,fg,bg);
        End;

    s := #201;
    FastWrite(x1,y1,s,fg,bg);
    s := #187;
    FastWrite(x2,y1,s,fg,bg);
    s := #200;
    FastWrite(x1,y2,s,fg,bg);
    s := #188;
    FastWrite(x2,y2,s,fg,bg);

```

```

End;
( * * * * * )
Procedure SetUpWindows;
Var
    i: Integer;
Begin
    New(ActiveWin);
    For i := 0 To MaxWin Do
        New(Windo[i]);

    With ActiveWin^ Do
        Begin
            WinX1 := 1;
            WinY1 := 1;
            Winx2 := 80;
            WinY2 := 25;
            With Scr Do
                Begin
                    CursX := WhereX;
                    CursY := WhereY;
                End;
            End;

        ActiveWin := Ptr(VidSeg, $0000);
        CurrentWindow := 0;
        With Windo[CurrentWindow]^ Do
            Begin
                WinX1 := 1;
                WinY1 := 1;
                Winx2 := 80;
                WinY2 := 25;
                With Scr Do
                    Begin
                        CursX := 1;
                        CursY := 1;
                    End;
                End;
            End;
        End;
    End;
( * * * * * )
Procedure OpenWindow;

```



```

Begin
If CurrentWindow < MaxWin Then
    Begin
        Windo[CurrentWindow]^ . Scr := ActiveWin^ . Scr;
        Windo[CurrentWindow]^ . Scr. CursX := WhereX;
        Windo[CurrentWindow]^ . Scr. CursY := WhereY;

CurrentWindow := CurrentWindow + 1;
With Windo[CurrentWindow]^ Do
    Begin
        WinX1 := CurrentWindow * 10;
        WinY1 := CurrentWindow * 2;
        WinX2 := WinX1 + 20;
        WinY2 := WinY1 + 5;
        With Scr Do
            Begin
                CursX := 1;
                CursY := 1;
            End;
            Window(WinX1, WinY1, WinX2, WinY2);
            PastBox(WinX1 - 1, WinY1 - 1, WinX2 + 1, WinY2 + 1, Yellow, Black);
            TextColor(Yellow)
            TextBackGround(Black);
            ClrScr;
        End
    End
End
End
( * * * * * )
Procedure CloseWindow;
Begin
If CurrentWindow > 0 Then
    Begin
        Windo[CurrentWindow]^ . Scr. CursX := WhereX;
        Windo[CurrentWindow]^ . Scr. CursY := WhereY;
        CurrentWindow := CurrentWindow - 1;
        ActivWin^ . Scr := Windo[CurrentWindow]^ . Scr;
        With Windo[CurrentWindow]^ do
            Begin
                Window(WinX1, WinY1, WinX2, WinY2);
                GotoXY(Scr. CursX, Scr. CursY);
            End
        End
    End
End

```

```

    End;
  End;
End;
( * * * * * )
Begin
  ClrScr;
  SetUpWindows;

  TextColor(Yellow+Blink);
  GotoXY(1,25);
  Write('Press any key to open windows...')
  GotoXY(1,1);
  TextColor(Yellow);

  FillWindow;
  For i:=1 To MaxWin Do
    Begin
      OpenWindow;
      FillWindow
    End;

  For i:=1 To MaxWin Do
    Begin
      CloseWindow;
      FillWindow;
    End
  End;
End;

```

当我们运行 Window Demo 时, CGA 用户会注意到屏幕上的雪花现象, 用 Turbo Pascal 编写代码这就是无法避免的, 但是用汇编语言写一个外部过程就可以消除雪花而且使屏幕更新更快。这里列出两个可以用作外部过程的汇编过程。第一个过程将一个逻辑屏幕传送到视频内存, 第二个过程将当前在视频内存中的屏幕传送到一个逻辑屏幕。在这两种情形, 雪花都已消除。要确保使用 Turbo Assembler 汇编这段程序。

```

.MODEL TPASCAL
.DATA
VIDTYPE DB?
.CODE
PUBLIC READSCR
PUBLIC WRITESCR
WRITESCR PROC FAR s:DWORD
;-----

```

;Save registers.

-----  
PUSH DS  
-----

;Get the monitor type.  
-----

XOR AX,AX ;Assign 0000h to ES

MOV ES,AX ;

MOV AX,0449h ;Assign offset of

MOV SI,AX ;video type location

MOV AX,ES:[SI] ;If video type is 7

CMP AL,7 ;then monitor is

JZ MONOL ;monochrome;

MOV AX,0B800H ;Load the color

MOV ES,AX ; segment into ES.

MOV VIDTYPE,I

JMP CONT1 ;Continue.

MONOL;

MOV AX,0B00H ;Load the monochrome

MOV ES,AX ; segment into ES.

MOV VIDTYPE,0

-----  
;Load buffer to screen.  
-----

CONT1 LDS SI,s ;Load buffer address in DS,SI

MOV DI,0 ;Point to start of memory.

MOV CX,2000 ;Move 2000 characters.

CLD ;Clear direction flag.

```

                MOV DX,3DAh           ;Load CRT port address.
NEXTCHAR1:      CMP VIDTYPE,0         ;If monochrome,don't check
                JZ   MOVECHAR1        ;for retrace.
WLOW1:          IN   AL,DX             ;Get CRT status.
                TEST AL,1             ;Is retrace off?
                JNZ  WLOW1             ; If off,wait for it to start.
                CLI                   ; No interrupts,please.

WHIGH1:         IN   AL,DX             ;Get CRT status.
                TEST AL,1             ; Is retrace on?
                JZ   WHIGH1            ;if on,wait for it to end.
MOV1CHAR1:      LODSW                 ;Get word from buffer to AX.
                STOSW                 ;Move word from AX to screen.
                STI                   ;Interrupts are allowed.
                LOOP NEXTCHAR1        ;Done yet?

```

ENDSTR1:

```

;-----
;Restore registers.
;-----

        pop    DS
        RET

```

WRITESCR ENDP

READSCR PROC FAR s;DWORD

```

;-----
;Save registers.
;-----

        PUSH   DS
;-----
;Get the monitor type

```

```

;-----
XOR  AX,AX                ;ASSIGN 0000H TO ES.
MOV  ES,AX

MOV  AX,0449h             ;Assign offset of
MOV  SI,AX                ;Video type location.

MOV  AX,ES:[SI]           ;If video type is 7
CMP  AL,7                 ;THEN MONITOR IS
LZ   MONO2                ;monochrome.
MOV  VIDTYPE,1

MOV  AX,0B800H            ;Load the color
MOV  DS,AX                ; segment into DS.

MONO2:
MOV  VIDTYPE,0
MOV  AX,0B000H            ;Load the monochrome
MOV  DS,AX                ; segment into DS.

CONT2: LES  DI,s           ;Load buffer address in ES;DI.
MOV  SI,0                 ;Point to start of memory.
MOV  CX,2000              ;Characters in screen.
CLD                       ;Clear direction flag.

;-----
;Transfer display memory to buffer.
;-----

MOV  DX,3DAh              ; Load CRT port address into
                           DX.
NEXTCHAR2 CMP  ES:VIDTYPE,0 ;If monochrome,don't check
JE   MOVECHAR2            ; for retrace.

WLOW2: IN  AL,DX           ;Get CRT status.

```

```

        TEST AL,1                ; Is retrace on?
        JZ     WHIGH2            ; If on, wait for it to end.
MOVECHAR2:LODSW                  ;Move word from acreen to AX.
        STOSW                    ;Move AX to buffer.
        LOOP NEXTCHAR2           ;Done yet?
        STI
ENDSTR2:

;-----
;Restore registers.
;-----

        POP    DS
        RET
READSCR  ENDP
CODE     ENDS
        END

```

刚才给出的程序清单包含两个过程，WriteScr 和 ReadScx，我们可以象下面这样在程序 Window Demo 中使用这两个过程：

```

( * $ R - , F + * )
Program WindowDemo;
Uses CRT;
Const
    MaxWin=5;
Type
    ScreenType=Record
        Pos:Array[1...80,1...25] of record
            Ch:Char;
            At :Byte;
        End;
        CursX;
        CursY:Byte;
    End;

```

```

WindowPtr = ^ WindowType;

```

```

WindowPtr=Record
  Scr:ScreenType;
  WinX1,
  WinX1,
  WinX2,
  WinY2:Byte;
End;

Var
  Ch:Char;
  i:Integer;
  Windo:Array[0..MaxWin] of WindowPtr;
  CurrentWindow:Integer;
  (***** )
Function VidSeg:Word;
Begin
  If Mem($0000:$0449)=7 Then
    VidSeg:$B000
  Else
    VidSeg:=$B800;
  End;
  (***** )
  {SI.FASTWRIT}
  Procedure FastWrite(x,y:Byte;
                     s:String;
                     bc,fc:Byte);External;

  {SI.FASTSCR}
  Procedure FastWriteScr(Var s:ScreenType);External;
  Procedure FastReadScr(Var s:ScreenType);External;
  (***** )
  Procedure FastBox(x1,y1,x2,y2,fg,bg,:Byte);
  Var
    i:Byte;
    s:String[i];
  Begin
    TextColor(fg);
    TextBackground(bg);

    s:=#205;
    For i:=(x1+1) to (x2-1) Do

```

```

    Begin
      FastWrite(i,y1,s,fg,bg);
      FastWrite(i,y2,s,fg,bg);
    End;

```

```

s := #186;
For i := (y1+1) to (y2-1) Do
  Begin
    FastWrite(i,x1,s,fg,bg);
    FastWrite(i,x2,s,fg,bg);
  End;

```

```

s := #201;
FastWrite(x1,y1,s,fg,bg);
s := #187;
FastWrite(x2,y1,s,fg,bg);
s := #200;
FastWrite(x1,y2,s,fg,bg);
s := #188;
FastWrite(x2,y2,s,fg,bg);
End;

```

```

( * * * * * )

```

```

Procedure SetUpWindows

```

```

Var

```

```

  i: Integer;

```

```

Begin

```

```

For i := 0 To MaxWin Do

```

```

  Begin

```

```

    New(Windo[i]);

```

```

    FillChar(Windo[i]^,Sizeof(Windo[i]^),0);

```

```

  END

```

```

  CurrentWindow := 0; With Windo[CurrentWindow]^ Do

```

```

    Begin

```

```

      WinX1 := 1;

```

```

      WinY1 := 1;

```

```

      Winx2 := 80;

```

```

      WinY2 := 25;

```

```

      With Scr Do

```

```

        Begin

```

```

          CursX := 1;

```



```

        CursY := 1;
    End;
End;
End
( * * * * * )
Procedure OpenWindow;
Begin
If CurrentWindow < MaxWin Then
    Begin
        Windo[CurrentWindow]^ . Scr := ActiveWin^ . Scr;
        Windo[CurrentWindow]^ . Scr. CursX := WhereX;
        Windo[CurrentWindow]^ . Scr. CursY := WhereY;

CurrentWindow := CurrentWindow + 1;
With Windo[CurrentWindow]^ Do
    Begin
        WinX1 := CurrentWindow * 10;
        WinY1 := CurrentWindow * 2;
        WinX2 := WinX1 + 20;
        WinY2 := WinY1 + 5;
        With Scr Do
            Begin
                CursX := 1;
                CursY := 1;
            End;
            Window(WinX1, WinY1, WinX2, WinY2);
            PastBox(WinX1 - 1, WinY1 - 1, WinX2 + 1, WinY2 + 1, Yellow, Black);
            TextColor(Yellow)
            TextBackGround(Black);
            ClrScr;
        End
    End
End
( * * * * * )
Procedure CloseWindow;
Begin
If CurrentWindow > 0 Then
    Begin
        Windo[CurrentWindow]^ . Scr. CursX := WhereX;
        Windo[CurrentWindow]^ . Scr. CursY := WhereY;
    End
End

```

```

    CurrentWindow := CurrentWindow - 1;
    ActivWin ^ . Scr := Windo[CurrentWindow] ^ . Scr;
    With Windo[CurrentWindow] ^ do
    Begin
        Window[WinX1, WinY1, WinX2, WinY2];
        GotoXY(Scr.CursX, Scr.CursY);
    End;
End;

( * * * * * )

Procedure FillWindow;
Var
    Ch: Char;
Begin
    Ch := ReadKey;
    Repeat
        Write(Chr(Random(80)+30));
        Delay(20);
    Until KeyPressed;
End;

( * * * * * )

Begin
    ClrScr;
    SetUpWindows;

    TextColor(Yellow+Blink);
    GotoXY(1, 25);
    Write('Press any key to open windows... ');
    GotoXY(1, 1);
    TextColor(Yellow);
    FillWindow;
    For i := 1 To MaxWin Do
        Begin
            OpenWindow;
            FillWindow;
        End;

    For i := 1 To MaxWin Do
        Begin
            CloseWindow;

```

```

    FillWindow;
End
End.
在这个程序中，删除了对变量 activeWin 的全部引用并用语句行
WriteScr(Windo[Current Window]^ .Scr);
代替
ActiveWin^ .Scr := Windo[Current Window]^ .Scr;
以及用
Read Scr(windo[Curren Window]^ .Scr);
来代替
Windo[Current Window]^ .Scr := ActiveWin^ .Scr;

```

这些快速而洁净的屏幕过程使我们的窗口看上去更具有专业的风格。而且现在我们可以把时间花在用各种有用信息和工具(日历、计算机、便笺等等)来填充窗口上了。

### § 10.3 Turbo Pascal 的图形单元(GRAPH Unit)

图形能力是个人计算机上的最强的特性之一，利用它可以建立图画、图表、多字体文本以及所有能够描画的东西，但是使用 PC 的图形方式的确要比文本方式复杂得多。为此，TurboPascal 专门提供了一个图形单元来简化用户的图形程序设计。GRAPH 单元是 Borland 提供的所有单元中最复杂的，它提供了超过 50 个过程以及 20 多个函数，另外还有一些常数，变量和数据类型。而且 Borland 包括了所有主要图形适配器的驱动程序和画出风格广泛的字符的字体程序。如果我们注重图形程序设计，那么单单 GRAPH 单元就值 Turbo Pascal 编译器的价。

#### § 10.3.1 描点

描点是一切图画的基础。描点实际上是在屏幕的指定位置上点亮一个像素。在图形方式下，我们可以在屏幕的任意位置上点亮像素，下面列出的程序就是在屏幕的随机指定的位置上点亮像素，它可以说明一些基本的图形程序设计技术

```

Program DemoPixel;
Uses CRT, GRAPH;
Var
    x, y,
    ErrorCode,
    GraphMode,
    GraphDriver: Integer;

Begin
    (* Initiate thr CGA high resolution mode. *)
    GraphDriver := CGA;
    GraphMode := CGAhi;

```

```

InitGraph(GraphDriver, GraphMode, D:\Tp5\
ErrorCode := GraphResult;
If ErrorCode() < grOK Then
    Begin
        WriteLn('Graphics error: ', GraphErrorMsg(ErrorCode));
        Halt;
        End;

    Repeat
        x := Random(640); (* CGA high resolution coordinates *)
        y := Random(200); (* 640 x 200 *)
        PutPixel(x, y, White);
        Delay(100);
    Until KeyPressed;

CloseGraph;
End.

```

程序由初始化 Turbo Pascal CGA 图形驱动程序开始。首先要指定图形适配器的类型和图形方式, 同时还要给出相应适配器的 .BGI 文件中包含了在特定适配器上画出合适的图形所需要的信息, 在本例中, 该文件在 D:\TPS 目录中。

调用 InitGraph 之后, 程序调用 GraphResult 函数以返回状态码。如果该码不等于 grOK, 我们就明白已经出现了错误。(grOK 是 GRAPH 单元中定义的常数, 其值为零。)如果检测到错误, 程序就将错误码传给 GraphErrorMsg 函数, 它可以返回描述出错条件的字符串。

```

ErrorCode := GraphResult;
If ErrorCode() < grOK Then
    Begin
        WriteLn('Graphics error: ', GraphErrorMsg(ErrorCode));
        Halt;
        End;

```

虽然检查错误不是必要的, 但却是一种好的想法。几乎每一个图形过程和函数如果用得不合适都可能产生严重的错误。

一般需要将 GraphResult 的值存放在一个变量中, 因为一旦此函数调过之后, 在下一错误发生之前它将返回 0 值。

如果我们知道所用的适配器, 那么在程序中规定显示适配器就很合适, 可是一旦我们错了, 程序也就无法运行了。注意屏幕的坐标限制(640 和 200)是作为固定码写进程序的, 如果利用计算机使用的图形适配器有不同的坐标系统, 程序是无法合适地运行的。为了使图形程序真正有用, 它必须能运行在任何图形适配器上, 而且它能尽可能地种用适配器的先进特性才是最理想的。

很幸运, TurboPascal 和 GRAPH 单元中提供了两个函数, GetMaxx 和 GetMaxy, 它们能

够告诉我们当前有效的图形适配器和图形方式下的最大 x 和 y 坐标, 这样, 经过一点修改, 前面的程序就可以运行在任何图形适配器上了。

经过修改更新的程序如下:

```
Program DemoPixel2;  
Uses CRT, GRAPH;  
Var  
    MaxX,  
    MaxY,  
    x,y,  
    ErrorCode,  
    GraphMode,  
    GraphDriver : Integer;  
  
GraphDriver := Detect;  
InitGraph(GraphDriver, GraphMode, D\TP5);  
ErrorCode := GraphResult;  
If ErrorCode <> grOK Then  
    Begin  
        WriteLn('Graphics Error: ', GraphErrorMsg(ErrorCode));  
        Halt;  
    End;  
  
MaxX := GetMaxX;  
MaxY := GetMaxy;  
  
    Repeat  
        X := Random(MaxX)+1;  
        Y := Random(MaxY)+1;  
        PutPixel(x,y,White);  
        Delay(100);  
    Until KeyPressed;  
  
CloseGraph;  
End.
```

这个程序虽然在功能上是简单的, 可是也说明了即使是一个简单的图形程序写起来也是相当复杂的。下面的章节将进一步说明如何利用 Turbo Pascal 图形单元提供的丰富的工具来进行更有意义的图形程序设计。

### § 10.3.2 画线

图形设计中的一个基本的任务是在两点间画一条直线。GRAPH 单元提供了 LINE 过程来做这项工作。不过你仍然会发现理解这样一个例程如何工作是有用处的，你也许会决定自己写一个特殊的图形例程。

画一条水平或垂直的线是容易的，你只需使一个坐标值保持固定不变，而沿着另一坐标描画像素点即可。可是你一旦决定画一条斜线，整个过程就变得复杂多了，你必须决定哪一些像素需要点亮，这要通过一个算法做，这个算法能通过给定一个坐标而计算出另一个坐标。

基于算法的图形系统几乎是所有图形设计的基础，由于算法不需与特定的比例因子相联系，因此它可以提供极大的柔软性(可适应性)，即你使用同样的算法可以画出小的方框又可以画出大的方框，或者用同样的算法既可的画出细线，也可以画出粗线。画直线的算法虽然有一点复杂却是直截了当的。

直线算法基于直线的代数方程：

$$Y = a + bX$$

这里 Y 是纵坐标(垂直坐标)，X 是横坐标(水平坐标)，a 是一个常量因子，b 是直线的斜率。一旦你决定了 a 和 b 的值，画直线是相当容易的。

为了计算 a 和 b，你需要两对坐标，这里用抽象代码表示成 x1:y1 和 x2:y2，这两对坐标指出图形屏幕上的两个点。

```
dx := (x2-x1);
```

```
dy := (y2-y1);
```

```
If dx <> 0 Then
```

```
  b := dy
```

```
  dx
```

```
Else
```

```
  b := 0;
```

```
  a := y1 - x1 * b;
```

变量 b 定义为用 X 坐标的差去除 Y 坐标的差。如果 X1 等于 X2，那么 b 定义为 0。一旦 b 被计算出来，a 只要用两对坐标之一就很容易被计算出来了。

有了 a 和 b 的定义，算法就完整了。为了画线，你只需沿着 x 轴，逐点计算相应的 y 坐标并描画像素即可。但这种方法也有一个问题，就是在 x 坐标范围与 y 坐标范围比相对较小时，直线会变得稀疏。为了补偿这一点，画线过程必须比较纵坐标间的距离和坐标间的距离。如果 y 坐标的间隙大于 x 坐标的间隙，移动应该沿着纵轴进行。

完整的画线过程包含在以下程序的 Plotline 过程中：

```
Program LineDemo;
```

```
Uses CRT, GRAPH;
```

```
Var
```

```
  MaxX,
```

```
  MaxY,
```

```
  ErrorCode,
```

```
  GraphMode;
```

GraphDriver : Integer;

( \* \* \* \* \* )

Procedure PlotLine(x1, y1, x2, y2, color : Integer);

Var

a, b : Real;

dx, dy,

x, yxi : Integer;

( \* \* \* \* \* )

Procedure Switch(var x, y : Integer);

Var

t : Word;

Begin

t := x;

x := y;

y := t;

End;

( \* \* \* \* \* )

Begin

If Abs(x1-x2) > Abs(y1-y2) then

Begin

(\* Gap between x's is greater than y's.

Trace horizontally \*)

If x1 > x2 then

Begin

Switch(x1, x2)

Switch(y1, y2);

End;

dx := (x2 - x1);

dy := (y2 - y1);

If dx < 0 Then

b := dy / dx

Else

```

    b := 0;
    a := y1 - x1 * b;
    For x := x1 to x2 do
        Begin
            y := Round(a + x * b);
            PutPixe(x, y, color);
        End;
    End;
Else
    Begin
        (* Gap between y's is greater than x's.
        Trace vertically. *)
        If y1 > y2 Then
            Begin
                Switch(x1, x2);
            End;

            dx := (x - x1);
            dy := (y - y1)

            If dx <> 0 Then
                b := dy / dx
            Else
                b := 0;
                a := y1 - x1 * b;

            For y := y1 to y2 do
                Begin
                    If b <> 0 Then
                        x := Round((y - a) / b)
                    Else
                        x := 0;
                    Putpixel(x, y, coulr);
                End;
            End;
        End;
    End;
    (* * * * * *)

```

```

    Begin

```



```

GraphDriver := Detect;
InitGraph(GraphDriver, GraphMode, /tP5);
ErrorCode := GraphResult;
If ErrorCode <> grOK Then
    Begin
        WriteLn('Graphics error: ', GraphErrorMsg(ErrorCode));
        Halt;
    End;

MaxX := GetMaxX;
MaxY := GetMaxY;

    Repeat
        PlotLine(Random(MaxX),
                    Random(MaxY),
                    Random(MaxX),
                    Random(MaxY),
                    White);
    Until KeyPressed;

CloseGraph;
End.

```

这个程序在计算机屏幕的两个随机点之间画直线(见图 10-3), 要停止该程序, 只需简单地按下任何一个键。

从上面的程序中可以看到画一条直线是不容易的, 因此也可以想象如果要写一个画多边形、椭圆、饼图或者更难画的文本字符的完整图形例程该有多么困难! GRAPH 单元对于所有图形程序员来说的确是极有价值的。

### § 10.3.3 圆、线和图形模式的综合使用

利用 GRAPH 单元的过程我们可以相对轻松地建立起极其复杂的图形, 能够控制其色彩、斜率和大小。Line 和 Circle 是两个最重要的过程, 最普通的图形, 如条形图, 圆饼图(百分率图)、直方图和多边形, 都是直线和圆的组合。

过程 Line 需要四个参数以坐标对形式提供。语句 Line (MaxX Div2, 0, MaxX Div2, MaxY) 将画出一条直线, 它垂直经过屏幕的中点, 从屏幕顶部直到屏幕底部。使用 MaxX Div2 来确定水平轴的中点可使语句独立于所用的坐标系统。

过程 Circle 需要三个参数, 前两个形成一个坐标对确定圆心的位置, 第三个参数则是以水平方向上的像素表示的圆的半径, Circle 将计算垂直方向上像素的个数以保持其合适的比例。垂直对水平的像素比率称为长宽比(Aspect ratio)。每个图形驱动程序都有一个长宽比用于确定图形图像的比例标尺。虽然 Turbo Pascal 允许我们更改长宽比, 但可能永远都不必这样做。

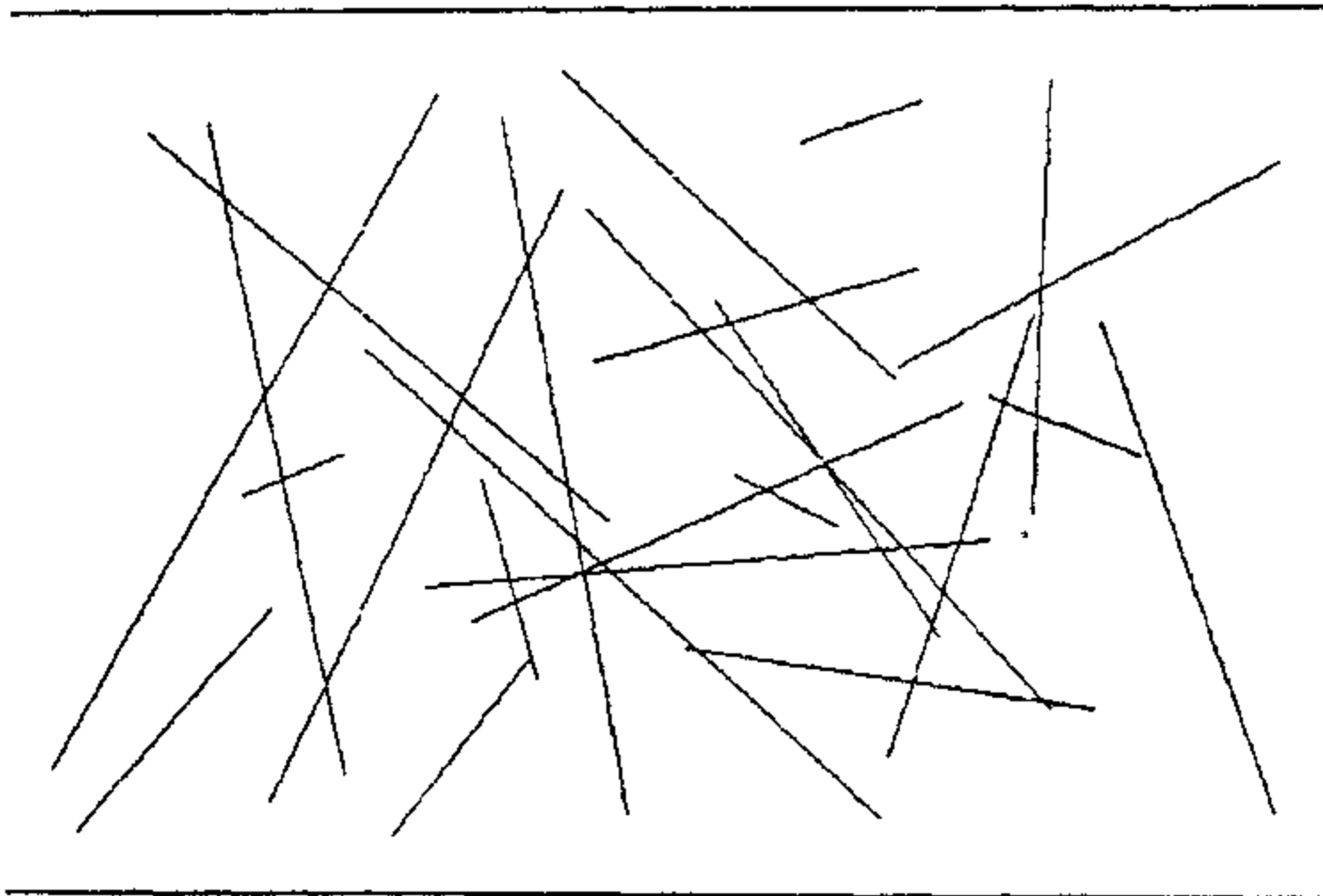


图 10-3 在图形方式下画线

现在回到一个例子上来，语句

```
Circle(MaxX Div 2, MaxY Div 2, 10);
```

将画出一个圆，它处于屏幕中心且具有 10 个水平像素大小的半径。我们也可以增加半径的大小直到圆的一部分已超出屏幕的边界，这时图象被削去一部分，以免写入内存时超出分配给图形形象的区域。

下面所列出的程序说明了 Line 和 circle 过程以及另外一些重要图形技术的使用方法。程序一开始在屏幕中心画交叉线，然后从屏幕边缘向中心画同心圆(参见图 10-4)，下一步程序保存一个环绕屏幕中心的矩形区域并用相反和正常的颜色显示它，最后，程序用随机选取的颜色和图形模式充填圆的各扇区。(参见图 10-5)

```
Program CircleDemo;
Uses CRT, GRAPH;
Var
    Palette : PaletteType;
    MaxX,
    MaxY,
    I,
    ErrorCode,
    GraphMode,
    GraphDriver : Integer;
    Size : Word;
    P : Pointer;
```

```

Begin
GraphDriver := Detect;
InitGraph(GraphDriver, GraphMode, D:\TP5);
ErrorCode := GraphResult;
If ErrorCode <> grOK Then
    Begin
        WriteLn('Graphics error: ', GraphErrorMsg(ErrorCode));
        Halt;
    End;
MaxX := GetMaxX;
MaxY := GetMaxY;

(* Draw lines on screen *)
Line(MaxX Div 2, 0, MaxX Div 2, MaxY);
Line(0, MaxY Div 2, MaxX, MaxY Div 2);
Line(0, 0, MaxX, MaxY);
Line(MaxX, 0, 0, MaxY);

(* Draw concentric circles *)
i := MaxY;
While i > 20 Do
    Begin
        Circle(MaxX Div 2, MaxY Div 2, i);
        i := i - 10;
    End;

(* Save a prtion of the screen *)
Size := ImageSize(Round(MaxX * 0.25),
                  Round(MaxY * 0.25),
                  Round(MaxY * 0.75));

GetMem(p, Size);
GetImage(Round(MaxX * 0.25),
          Round(MaxY * 0.25),
          Round(MaxY * 0.75), p);

(* Flash a portion of the screen *)
For i := 1 to 6 Do
    Begin
        PutImage(Round(MaxX * 0.25),

```

```

        Round (MaxY * 0.25),
        P ^ , NotPut);

End;

GetPalette(Palette);

(* Fill in portions of the graphic image *)
Repeat
    SetFillStyle(Random(9),Random(Palette.Size)+1);

    FloodFill(Random(MaxX),Random(MaxY),White);
Until KeyPressed;

ReadLn;
CloseGraph;
End.

```

在刚刚给出的程序例子中说明了如何保存屏幕图形的一部分并以另一种形式重显它。保存一个图形影象的过程需要三步：

1. 确定为了存贮图形影象将需要多少内存。
2. 将图象保存在这个缓冲区内。

这个过程用以下代码实现：

```

Size := ImageSize(Round (MaxX * 0.25),
                  Round (MaxY * 0.25),
                  Round (MaxX * 0.25),
                  Round (MaxY * 0.75);
GetMem(P,Size);
GetImage(Round (MaxX * 0.25),
         Rond (MaxY * 0.25),
         Round (MaxX * 0.75),

         Round (MaxY * 0.75), P ^ );

```

第一个语句用 ImageSize 函数计算存贮由两个坐标对定义的矩形图象所需要的内存数量，这个内存量与所有显示适配器和图形方式有关，但不能超出 64k。然后 GetMem 过程从堆中分配内存给 Pointer 变量 P，最后，GetImage 过程将坐标对定义的图象送到 P 所指向的缓冲区中。

一旦我们保存了一个图形影象，就可以随时检索它了。在我们的例子程序中，这是由过程 PutImage 完成的。

```
PutImage(Round (MaxX * 0.25),
```

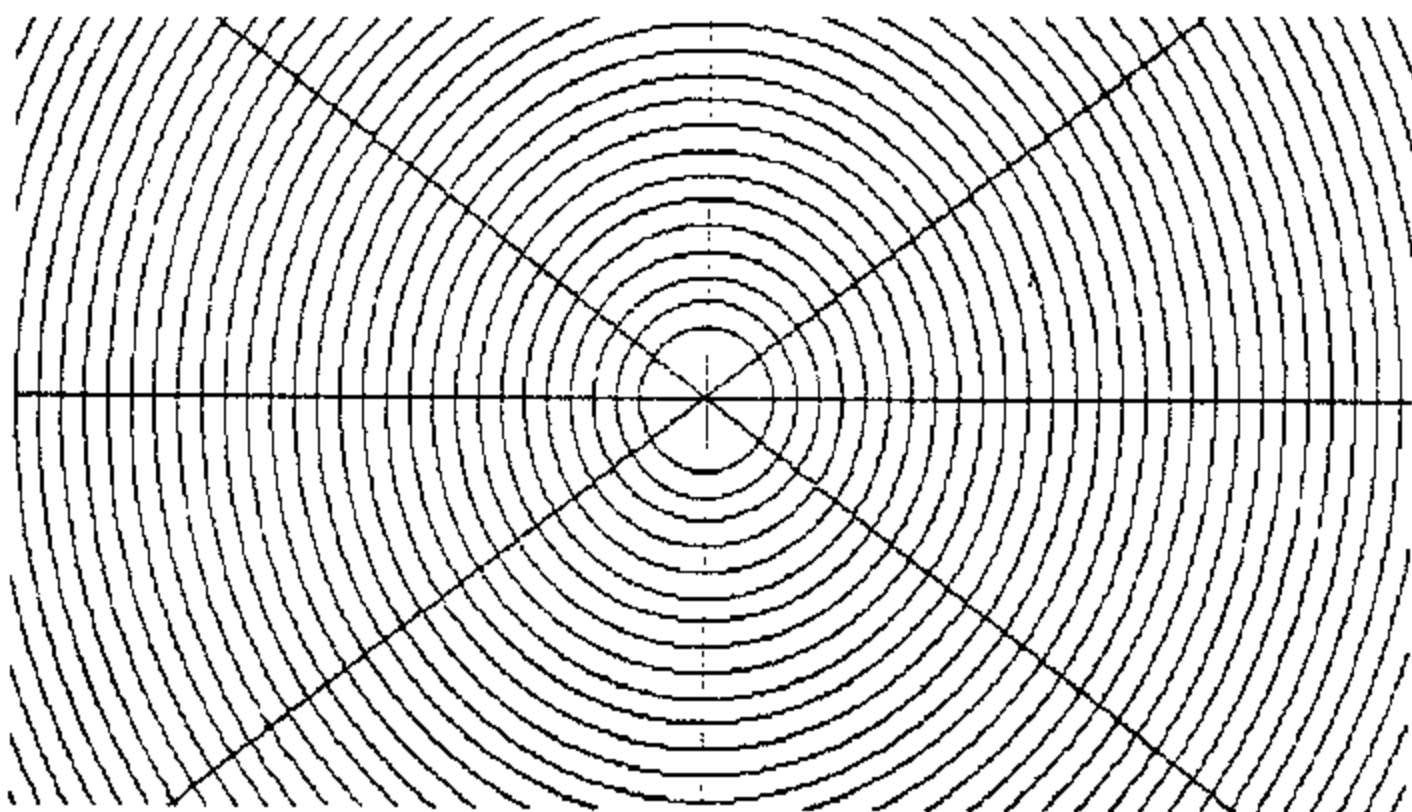


图 10-4 画线与圆

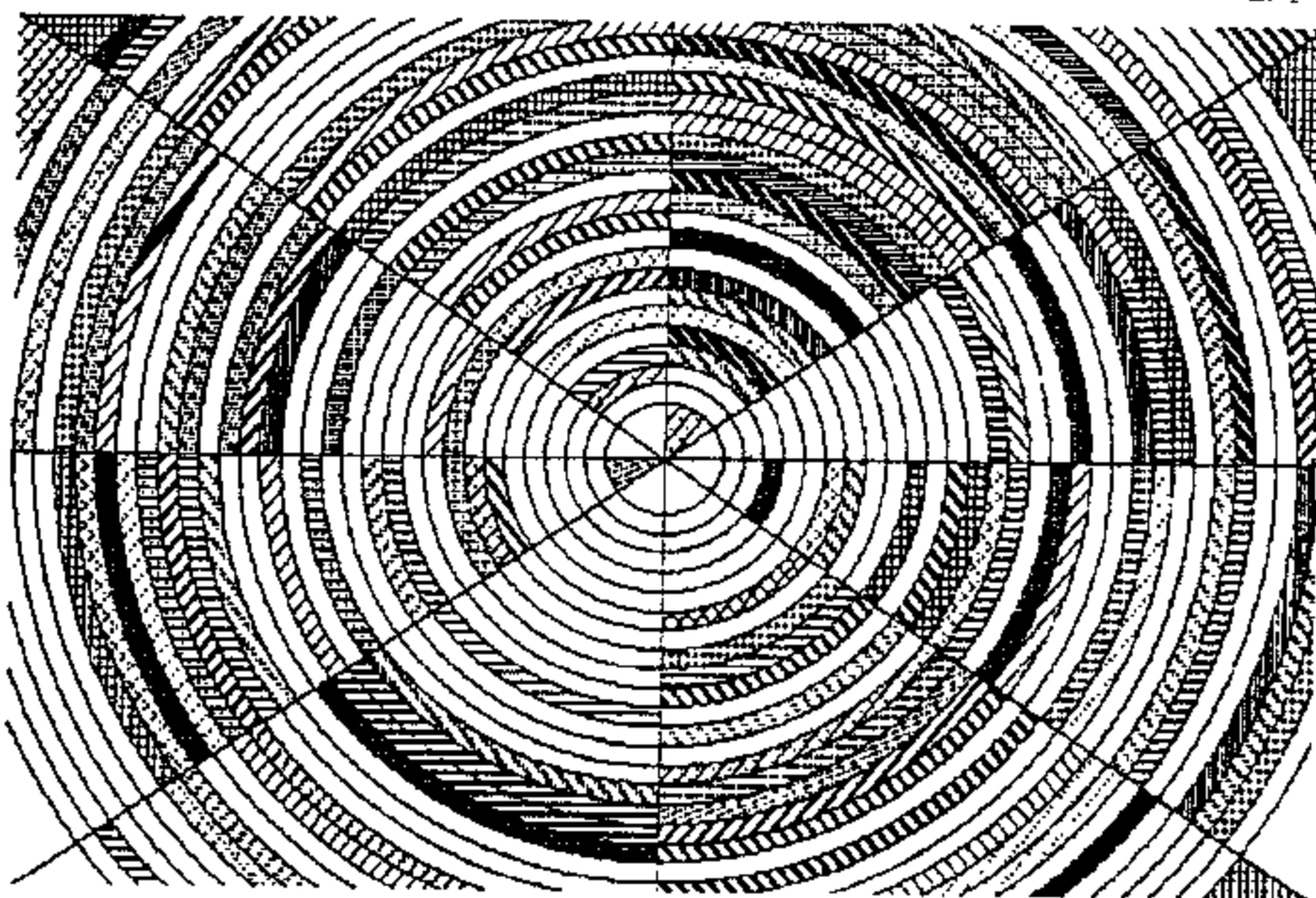


图 10-5 充填颜色和图形模式

```
Roun (MaxY * 0.25),  
P^, NotPut);
```

PutImage 需要四个参数，前两个是一个坐标对，指出所保存图象的左上角应在屏幕上出现的位置。第三个参数是一个指针，指向图象所存贮的缓冲区。第四个参数最有趣，因为它确定图象如何显现。上述例子中使用了常数 NotPut，它用一个按位取反过程（每位的值都取反），使图象的颜色反转，产生一个视频反显的效果。

程序的最后一部分是用不同的颜色和图形模式来充填图形。这个过程由 GetPalette 过程

开始,它返回有关当前图形驱动程序和方式的彩色能力的信息。

```
Get Palette(palette);
```

( \* 充填图象的各部分 \* )

```
Repeat
```

```
SetFillStyle(Random(9), Random(Palette,Size)+1);
```

```
FloodFill(Random(MaxX)< Random(MaxY), White);
```

```
Until KeyPressed;
```

过程 getPalette 需要一个类型为 PaletteType 的参数, PaletteType 的定义如下:

```
type
```

```
PaletteType = Record
```

```
    size : Byte;
```

```
    Color : Array[0..MaxColors] of ShortInt;
```

Size 域包含当前可用颜色的种数,而 Color 中包含对应颜色的数值编码。在用一个对 Get-Palette 的调用我们就可以准确地了解到我们必须在哪些图形颜色下工作。

完成困难工作的过程是 SetFillStyle 和 floodFill,它们要用各种各样的颜色描画不同的图形模式。GRAPH 单元中定义了 12 种不同的图形模式用于充填图形。SetFillStyle 假定我们选择了想用的图形模式以及想用米显示的颜色, FloodFill 使用你用 SetFill Style 定义的模式和颜色“喷涂”多边形或圆的内部。为了做到这一点, FloodFill 需要两方面的信息:(1)需要充填区域内部的象素的位置,(2)该区域边缘的颜色。边缘颜色是 FloodFill 能确定何处该停止充填的唯一手段。

PutImage 过程在前面例子中被用来产生视颠反显的效果,它也能用来在屏幕上“拖曳”图象。拖曳图象的重要价值在于图象移动时并不干扰图象“下面”的象素。任何使用过桌面印刷或其它“喷涂上色”程序的人对拖曳概念都是熟悉的。

为了拉动图象,你必须做两件事 第一,被拖曳的图象必须存放在与基础图象分离的缓冲区域里,第二,你必须用 XORPut 作为第四个参数使用 putImage 过程。下面的样板程序说明了这是怎么做的。它保存一个圆的图象,再往屏幕上画线,然后围绕屏幕拖曳这个圆。(参见图 10-6)。

```
Program CircleDemo2;
```

```
Uses CRT, GRAPH;Var
```

```
    Direction : (up,down,ritht,left);
```

```
    XX,YY,
```

```
    MaxX,
```

```
    MaxY,
```

```
    ErrorCode,
```

```
    GraphMode,
```

```
    GraphDriver : Integer;
```

```
    Size : Word;
```

```
    P : Pointer;
```

```

Begin
  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, D:\TP5);
  ErrorCode := GraphResult;
  If ErrorCode <> grOK Then
    Begin
      WriteLn('Graphics error: ', GraphErrorMsg(ErrorCode));
      Halt;
    End;

  MaxX := GetMaxX;
  MaxY := GetMaxY;

  (* Draw a circle, save it, and clear the screen. *)

  Circle(MaxX Div 2, MaxY Div 2, 20);

  XX := Round(MaxX * 0.45);
  YY := Round(MaxY * 0.45);
  Size := ImageSize(XX, YY,
                    Round(MaxX * 0.55),
                    Round(MaxY * 0.55));

  GetMem(P, Size);
  GetImage(XX, YY,
           Round(MaxX * 0.55),
           Round(MaxY * 0.55), P^);

  ClearViewPort;

  (* Draw lines on screen *)
  Line(MaxX Div 2, 0, MaxX Div 2, MaxY);
  Line(0, MaxY Div 2, MaxX, MaxY Div 2);
  Line(0, 0, MaxX, MaxY);
  Line(MaxX, 0, 0, MaxY);

  (* Start dragging the circle. *)
  Direction := down;
  PutImage(XX, YY, P^, XORPut);

```

```
Repeat  
PutImage(XX, YY, P ^ ,XORPut);
```

```
Case Direction Of
```

```
Down ;
```

```
Begin
```

```
YY := YY + 5;
```

```
If YY > (MaxY * 0.75) Then
```

```
Direction := Left;
```

```
End;
```

```
Left;
```

```
Begin
```

```
XX := XX - 5;
```

```
If XX < (MaxX * 0.25) Then
```

```
Direction := Up;
```

```
End;
```

```
Up ;
```

```
Begin
```

```
YY := YY - 5;
```

```
If YY < (MaxY * 0.25) Then
```

```
Direction := Right;
```

```
End;
```

```
Right ;
```

```
Begin
```

```
XX := XX + 5;
```

```
If XX > (MaxX * 0.75) Then
```

```
Direction := Down;
```

```
End;
```

```
End;
```

```
PutImage(XX, YY, P ^ ,XORPut);
```

```
Until KeyPressed;
```

```
CloseGraph;
```

```
End.
```



拖 是用语句

```
PutImage(XX, YY, P^, XORPut);
```

来完成的。这条语句在移动的每一步中使用两次，第一次用它时 PutImage 擦去圆，第二次它在一个新的位置上重画这个圆。随着圆绕着屏幕移动，直线保持不受干扰。使用这个技术的重要的一点是设法定义存贮被拖 图象的可能的最小缓冲区。缓冲区越大，更新图象所用的时间就越长，拖曳 就显得更慢。

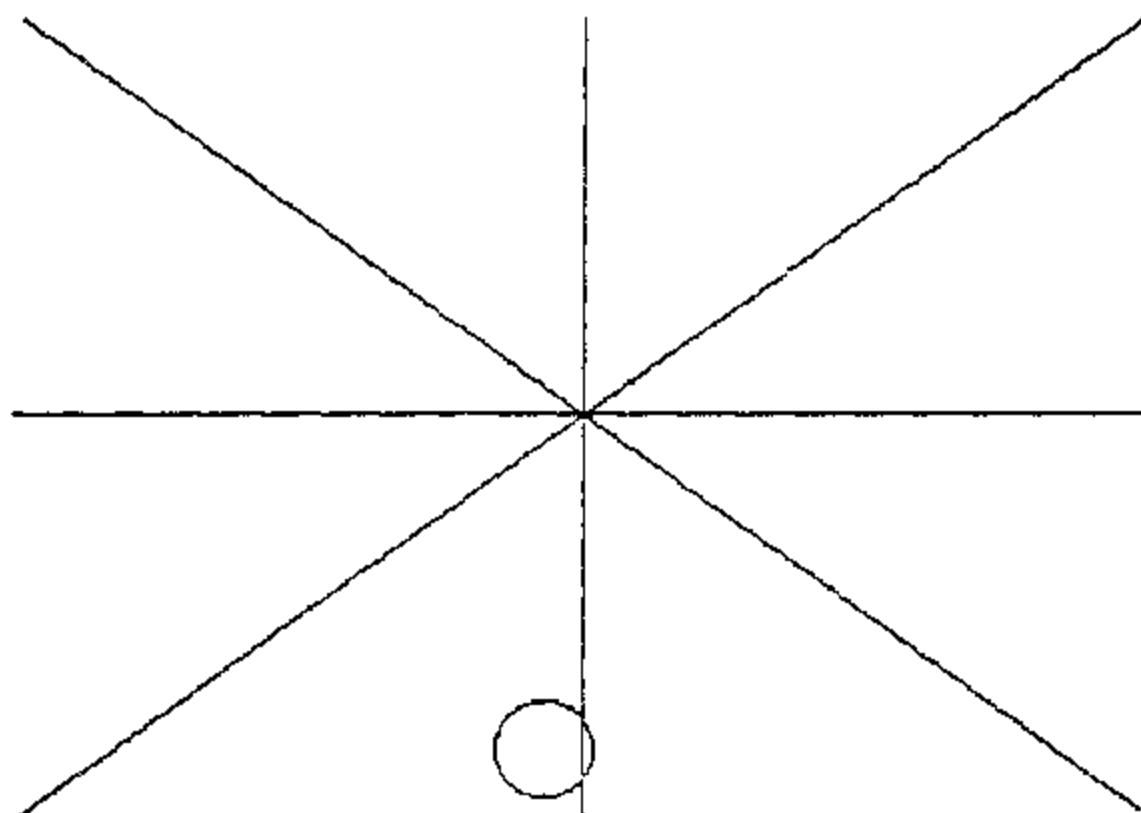


图 10-6 在图形方式 | 拖曳 一个圆

#### § 10.3.4 图形文本

图形中不使用正文就不会非常有用，综合文本与图形的能力是桌面印刷工业的基础。在 GRAPH 单元中包含了五种字体：一种位映象的字体和四种“笔划”字体。位映象字体具有固定数目的元素，当它放大时，位映象字符会变得很粗糙，因为它们的定义无法改变成与较大的尺寸相匹配，(参见图 10-7)。

笔划字体是通过一个算法来建立字符的，因为算法独立于大小规模，笔划字体随着放大实际上变得更加精密。(参见图 10-8)。

下面所列的程序说明了位映象和笔划字体的不同。它显示五种 GRAPH 字体的每一种，开始是缺省的位映象字体，逐渐增加大小尺寸。

```
Program TextDemo;
```

```
Uses CRT, GRAPH;
```

```
Const
```

```
CharType : Array [0..4] of String[20] =
```

```
( 'Default', 'Triplex', 'Small', 'Sans Serif', 'Gothic' );
```

Var

S : String;  
MaxX,  
MaxY,  
i,j,k,  
ErrorCode,  
GraphMode,  
GraphDriver : Integer;

Begin

GraphDriver := Detect;  
InitGraph(GraphDriver,GraphMode,'D:/TP5');  
GraphMode := GetMaxMode;  
InitGraph(GraphDriver,GraphMode,'D:/TP5');  
ErrorCode := GraphResult;  
If ErrorCode <> grOK Then  
    Begin  
WriteLn('Graphics error: ',GraphErrorMsg(ErrorCode));  
    Halt;  
    End;

MaxX := GetMaxX;  
MaxY := GetMaxY;

For i := 0 to 4 Do

    Begin  
    K := 0;  
    For j := 1 To 10 Do  
        SetTextStyle(i,HorizDir,j);  
        OutTextXY(j\*20,k,CharType[i]);  
        K := K + TextHeight(CharType[i]) + 10;  
    End;

    SetTextStyle(0,HorizDir,2);  
    S := 'Press ENTER...';  
    OutTextXY(MaxX - TextWidth(S),MaxY - TextHeight(S),S);  
    ReadLn;  
    ClearDevice;  
    End;

CloseGraph;

End.

Default

Default

Default

Default

Default

Default

Default

number 10

Press ENTER...

图 10-7 位映象字体

为了选择字体要使用 `SetTextStyle` 过程。该过程用到三个参数，第一个参数是字体，选择范围是 0(缺省)到 4(哥德体)；第二个参数是所写文本的方向，你可以有两种选择：0(水平)或 1(垂直)；第三个参数控制字符的大小。正常字符的尺寸对于缺省字体为 1，对于笔划字体为 4，但它的都能增加到正常尺寸的十倍。

一旦我们选择了字体，就可以用 `OutTextXY` 过程来显示文本，我们必须提供显示字符串的起始坐标作为参数。注意到在程序例子中，`OutTextXY` 是与函数 `TextHeight` 联合使用的，它可以判定下一行正文应在何处显示。`textHeight` 是有必要的，因为每一种字体都有自己的比例标尺。如果我们想放一行文本到另一行之下，就必须确定当前文本的高度并计算为了写下一行文本而必须下移的距离。在程序中，语句。

```
K := K + text Height(Char Type[i]) + 10;
```

向下移动下一行文本到当前行尺寸加上 10 个象

素。正如我们看到的，即使已经为我们提供了字体定义，在图形方式下使用文本也并不容易。

### § 10.3.5 多边形及填彩

为了圆满结束对 Turbo Pascal 图形的讨论，让我们再考虑一个多边形描画及填彩的例子。多边形的建立由要过程 `fillPoly` 完成，该过程可以建立任意边数的多边形并使用 `SetFillStyle` 过程定义的颜色和图形模式将其填彩。

为了使用 `FillPoly` 过程，首先必须定义 `PointType` 的变量数组。`Point Type` 是在 `GRAPH`

Triplex  
Triplex  
Triplex  
Triplex  
Triplex  
Triplex  
Triplex  
Triplex

Press ENTER...

图 10-8 笔划字体

单元中定义的记录类型, 包含 X 和 Y 坐标。

下面的程序例子给出这是怎么做的。

```

Program ColorDemo;
Uses CRT, GRAPH;
Type
    TriType = Array[1..3] of PointType;
Var
    Tri: TTriType;
    S: String;    MaxX,
    MaxY,
    i, j, k,
    ErrorCode,
    GraphMode,
    GraphDriver : Integer;

    Palette: PaletteType;
Begin
    GraphDriver := Detect;
    InitGraph(GraphDriver, GraphMode, 'D:/TP5');
    GraphMode := GetMaxMode;
    InitGraph(GraphDriver, GraphMode, 'D:/TP5');
    ErrorCode := GraphResult;

```

```

If ErrorCode <> grOK Then
    Begin
        WriteLn( 'Graphics error: ', GraphErrorMsg(ErrorCode));
        Halt;
        End;

MaxX := GetMaxX;
MaxY := GetMaxY;

GetPalette(Palette);
SEtFillStyle(3,3);
While Not Keypressed Do
    Begin
        SetFillStyle(Random(13),Random(Palette.Size)+1);
        Tri[1].X:=Random(MaxX);
        Tri[1].Y:=Random(MaxY);
        Tri[2].X:=Random(MaxX);
        Tri[2].Y:=Random(MaxY);
        Tri[3].X:=Random(MaxX);
        Tri[3].Y:=Random(MaxY);
        FillPoly(3,Tri);
        Delay(100);
        x1:=x1+(MaxX Div 12);
        y1:=y1+(MaxY Div 12);
    EndLn;
CloseGraph;
End.

```

程序例子使用 FillPoly 过程画三角形。因为三角形可以由三个点来完全定义，因此下列定义是充分的：

```

Type
TnType = Array [1..3] of Point Type;

```

命名为 Tri 的变量说明成 TnType。为了建立一个三角形，程序首先在 Tri 中定义了三个坐标对。对 FillPoly 的调用要传递多边形的顶点数(本例中是三)以及变量 Tri，它包含了坐标。正如下面所给出的，三角形的坐标是随机确定的。

```

Tri[1].X := Randow(MaxX);
Tri[1].Y := randow(MaxY);
Tri[2].X := Randow(MaxX);
Tri[2].Y := ranodw(MaxY);
Tri[3].X := Randow(Max X);
Tro[3].Y := Randow(MaxY);

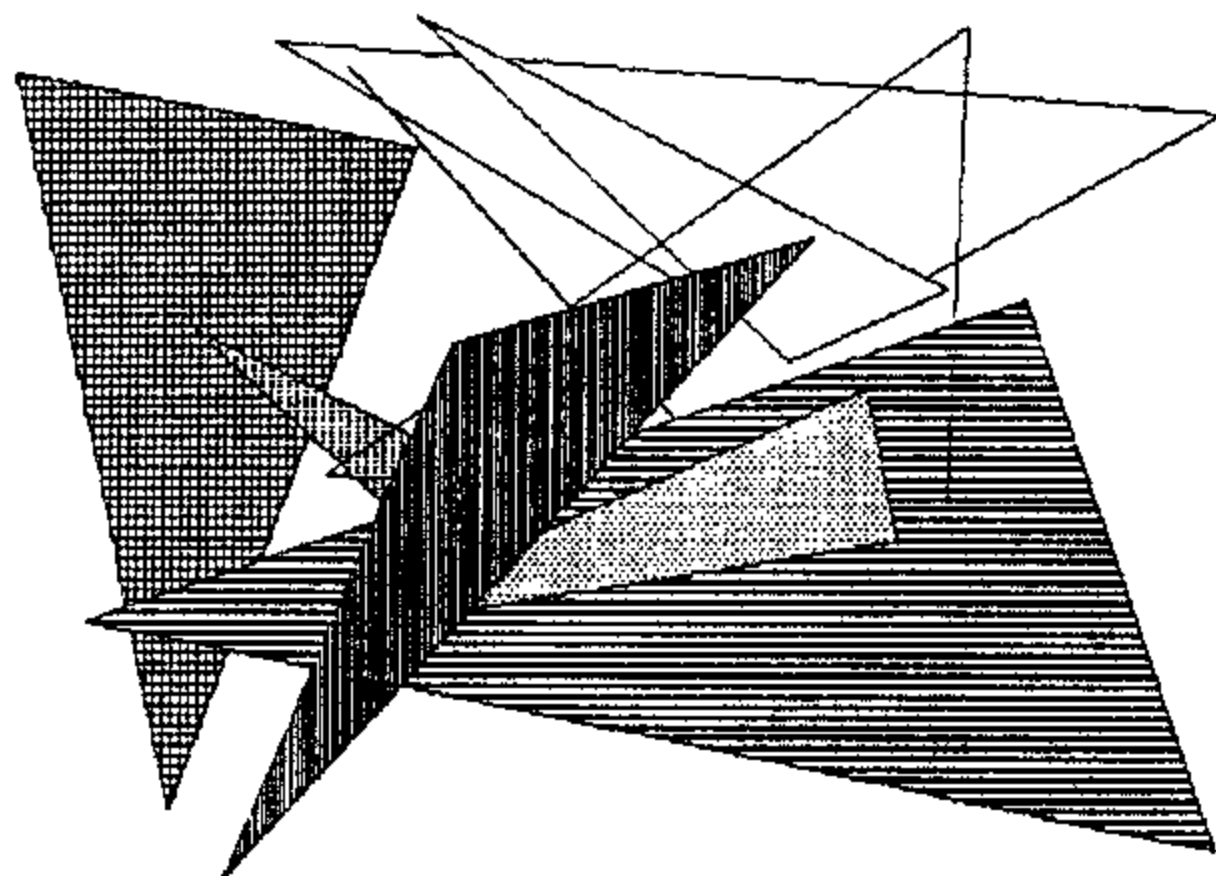
```

```
FillPoly(3, Tri);
```

对于每个画好的三角形，颜色和图形模式也是用 `setFillStyle` 随机选择的，如下所示：

```
SetFillStyle(Random(13), Random(Palette.Size)+1);
```

---



---

图 10-9 用颜色和图形模式充填三角形

这条语句随机地选取了 GRAPH 单元中定义的 12 种充填图形模式之一，也随机地从视频方式调色板中选取一种颜色(参见图 10-9)。这样，当我们运行这个程序时就可以看到随机产生的三角形并具随机选择的图形模式和颜色。

PC 的图形潜力是巨大的，但开发它则对于最有经验的程序员也仍是一个挑战。对于初学者 Turbo Pascal 的 GRAPH 单元提供了学习图形程序设计的一种轻松的途径。对于专业程序员，GRAPH 单元的丰富的例程集和数据结构提供了建立复杂应用的基本构件。

# 第十一章 DOS:软中断和硬中断

磁盘操作系统(DOS:Disk Operating System)实际上是一组软件例程,这些例程管理控制着个人计算机的各种各样的物理设备(键盘、显示器、磁盘驱动器、打印机等等),使数据能够无差错地送达合适的位置。

DOS 可以在 Intel808/8088 系列微处理机上运行,它可以看作是由四个基本层组成的体系结构。紧贴着硬件的是基本输入/输出系统(BIOS: Basic Input Output System),该层基于 ROM,并直接与硬件通讯。再外一层是 DOS 内核,该层作为操作系统的一部分,与应用程序交互,提供系统调用,本书中提到的 DOS 服务,指的就是 DOS 内核层提供的系统调用服务。由于有 BIOS 的支持,DOS 内核并不依赖于硬件。再外一层是命令处理器,是 DOS 的基本用户界面,它利用一个称为 COMMAND.COM 的文件来处理 DOS 命令。最外一层是应用层,通常是一组用户开发的软件,如数据库管理系统、字处理软件、以及象 Turbo Pascal 这样的软件。

作为程序员,主要关心是内二层,即 BIOS 和 DOS 内核层。当然我们也可以认为因为不知道 BIOS 和 DOS 服务,仅用 Turbo Pascal 编写程序已经工作得很好,但是这是没有意识到 Turbo Pascal 一直在使用 BIOS 和 DOS 服务完成诸如写一个磁盘文件、在显示器上显示信息、获得当前日期和时间等任务。此外,还有两个理由说明为什么我们应该了解 DOS 和 BIOS 的服务并掌握如何直接使用它们。

首先,Turbo Pascal 虽然利用了其中的许多服务,但并没有使用其全部能力。如果我们想完全控制自己的 PC 机,就不得不学习驾驭 DOS 和 BIOS 服务的能力。其次,即使从不需要使用这些服务,学习有关这些服务的内容也将大大增进我们对个人计算机和操作系统的理解。

## § 11.1 DOS 与 BIOS 服务

BIOS 由一组 I/O 例程组成,用来直接与处理器硬件和外部设备通讯,它随着机器型号的不同而不同,它由计算机制造厂家提供。如果一厂家宣称其机器与 IBM PC 兼容,则它提供了 BIOS 级一致的功能接口,从而使机器硬件的差异对于用户的以及利用 BIOS 实现其功能的应用程序来说是透明的。

BIOS 层含有与缺省配置硬件相关的设备驱动模块。这些设备包括:

- 显示器和键盘(CON)
- 宽行打印机(PRN)
- 辅助设备(AUX)
- 系统时钟(CLOCK)
- 磁盘驱动(块设备)

DOS 内核利用 I/O 请求包来与这些设备驱动模块通讯。这些 I/O 请求由相应的驱动模块转换为针对特定设备控制硬件的命令。

BIOS 对设备的控制和访问是通过中断机制来完成的。一般每一设备均与一个 BIOS 中断号和存储在 ROM 中的该设备的专用处理程序相联系, 中断处理程序的入口存放在一块称为中断向量表的特殊内存区域中。关于中断机制, 我们在下一节将详细讨论。利用中断机制, 程序员在使用处理程序时, 就不必知道并记住相应的入口地址。除了利用 BIOS 例程之外, 还可以用其它方法访问设备, 但 DOS 是利用 BIOS 来访问设备的。

DOS 层是指 DOS 的最核心部分, 不包括命令解释器。DOS 层实现了对应用程序的管理, 还向用户提供了一套独立于硬件的系统功能, 包括:

- 文件和记录的管理
- 内存管理
- 字符设备 I/O
- “假脱机”(Spooling)
- 系统时钟的访问

DOS 内核通过 DOS 功能调用提供对应用程序的服务。DOS 的功能调用也是利用中断机制向外提供系统功能接口。

一个应用程序既可以直接利用 BIOS 也可以利用 DOS 功能调用来访问一个设备。应用程序也可以不用 DOS 或 BIOS 而直接访问和控制设备。这类程序通常是一些设备驱动程序以及其它一些功能要求较为特殊的程序。

应用程序调用 DOS 功能或利用 BIOS, 首先都要将寄存器设置为指定的参数, 然后再通过操作系统来完成调用(软中断)。

Intel 8086/8088 系列的微处理器(也包括 80286 等)包含标准的寄存器组, 共 14 个寄存器, 每个寄存器 16 位长, 这就是为什么 8086/8088 被称为 16 位微处理器。在 Turbo Pascal 中, 16 位存储块叫做一个字。8086/8088 的寄存器如图 11-1 所示。

前四个寄存器—AX, BX, CX 和 DX—是在计算、比较和其它操作中用来临时存放数据的通用区域。汇编语言程序员使用这些寄存器的方式就象 Pascal 程序员使用变量一样。每个通用寄存器都能分成两个一字节的寄存器, 这样 AX 就由 AH 和 AL 组成, 同样 BX 与 BH、BL; CX 与 CH、CL; DX 与 DH、DL 都有这种关系。通用寄存器是 DOS 调用和 BIOS 服务最常用的寄存器。

8086/8088 还有四个段寄存器: CS、DS、SS 和 ES。CS 用于程序代码段, DS 用于数据段, SS 用于栈段, ES 用于特殊操作的临时段(附加的备用段)。CS 和 SS 寄存器存有一旦改变就会危及程序完整性的关键数据, 因此 Turbo Pascal 不允许我们在做 DOS 或 BIOS 调用时存取这些寄存器, 但 DS 和 ES 偶尔用于传递段地址。

一个内存地址由一段和一个偏移量(相对地址)组成, 8086/8088 包含五个偏移量寄存器: IP, SP, BP, SI 和 DI。它们用于段寄存器拼合以寻址内存的特定位置(内存单元)。Turbo Pascal 只允许存取 SI、DI 和 BP, 而 IP 和 SP 决不在 DOS 或 BIOS 调用中使用。

最后要说明的是标志寄存器, 它包含有关最近执行指令的状态信息。标志字节中不同的位指明 CPU 操作结果的特定条件, 但没有用完所有位。标志寄存器主要用来标识错误条件。虽然它能在 Turbo Pascal 中使用, 但一般对 DOS 或 BIOS 调用来说不是必需的, 它们通常在一个通用寄存器中返回错误码。



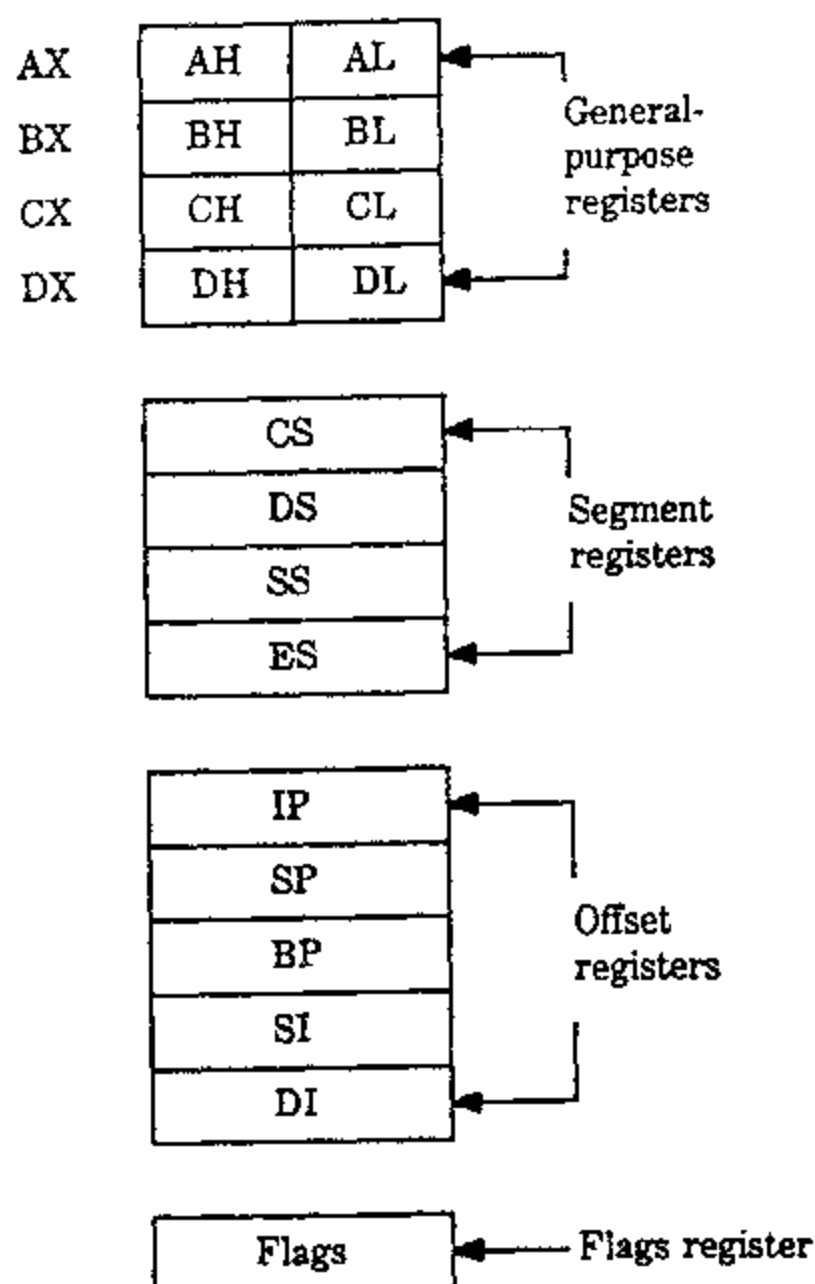


图 11-1 8086/8088 寄存器

## § 11.2 DOS 与中断

上节已经提到 BIOS 或 DOS 服务均通过中断机制执行，那么什么是中断？中断从概念上很容易理解。生活中就充满了中断，如果我们有一部电话，那么当电话铃响时，我们就要停下手头的工作，接电话，然后在我们交谈结束后恢复手头的工作。在某种表义上，计算机在微处理器内部也有一个小小的电话铃使它能停下来做一会其它事。

中断是计算机操作的其础。如果我们想开发个人计算机的全部能力，我们就必须理解中断。

当我们的电话铃响时我们是不是偶尔也会感到不便？但我们想象一下如果一个电话机不会振铃该会多么不便！我们将不得不一次又一次地拿起听筒看看是否有谁打电话给我们。这不仅浪费时间，而且我们将冒丢失电话呼叫的风险，如果在我们拿起听筒这前有人打电话给我们又挂掉的话。

当然，我们的电话是会振铃的，因此我们可以放心地做其它事，而仅当电话铃响我们知道有人呼叫时才回答电话。与此相类似，中断(机制)允许计算机工作直到某个件发生需要计算机关注时为止。

采用“中断”是提高计算机操作效率的重要因素，因为通常计算机必须对许多微处理器外部的事件作出响应，例如我们在键盘上打字，它会响应。如果经常不断地去查看有什么外部

事件要处理，就会化去计算机大量的时间。这在早期有些计算机上是这么做的。叫做“定时询问技术”，现在有了中断机制，处理器就不必浪费时间仅仅为了寻找工作，一旦有工作要做时，这项工作自己就来找处理器了。

中断的例子是很多的，按动键盘上的按键就是一个。另一个重要的中断是个人计算机内部时钟的报时信号，这个中断每秒大约要打断我们的微处理器 18 次，它请求增加 DOS 的时间和日期。平时我们注意不到它，因为它发生得太快了。再一个例子是软盘驱动器产生中断，通知我们的微处理器，该软盘的某个操作已经全部结束了。事实上，中断在所有时间里都发生，但我们注意不到它们，因为它们通常只要微处理器做非常少的工作。

中断有两种类型：硬件中断和软件中断。我们前面所举的例子都是硬件中断，如按下一个键，系统时钟的滴答跳动，数据进入串行口等等。硬件中断是由要求微处理器关注的设备动作引起，在计算机电路上产生并由一个特殊芯片 8259 中断控制器控制的中断。当一个硬件中断发生时，8259 芯片扮演着交通警察的角色，确保中断沿着正确的方向前进。8259 接收一个中断请求，评估其优先级，再转送这个请求给它需要的过程。

软件中断由程序请求特定 BIOS 和 DOS 服务引起。在 Turbo pascal 中，Intr 命令和 MS-DOS 命令产生软件中断。然而，无论硬件中断还是软件中断，所有中断都使用中断向量表 (interrupt vector table)。

中断向量表是地址位于 PC 内存最低部分的一个内存数组。这个数组有 1024 字节长，包含了所有由中断触发的例程的地址。因为一个地址需要 4 个字节 (段地址和偏移量地址，1024 字节的中断向量表最多共能容纳 256 个中断地址。

一个中断发生后，它就从中断向量表中取出一个地址，跳转到相应的内存单元，然后执行定位在那儿的例行程序。中断向量表中的每个地址都只供一个中断使用，例如，中断 8—时钟计数器—总是取出在偏移量为 0020h 处的地址。

中断类型从 0 到 FFH 编号。从 0 到 0FH 的中断是低级中断，全部给内部

外部硬件中断使用；从 10h 到 1Fh 的中断由 BIOS 使用；从 20h 到 3Fh 的中断由 DOS 作为其软件中断。表中余下的中断供用户或外部硬件设备和系统设备驱动程序使用。表 11—1，表 11—2 和表 11—3 列出了一些硬件和软件中断。

中断号	类别	偏移地址	目的描述
00	内部硬件	0000h	被零除
01	内部硬件	0004h	单步
02	内部硬件	0008h	NMI(不可屏蔽中断)
03	内部软件	000Ch	断点陷井 (供调试器用)
04	内部硬件	0010h	溢出
(05)	ROM—BIOS 输件	0014h	打印屏幕
06	内部硬件	0018h	保留
07	内部硬件	001Ch	保留

08	外部硬件	0020h	时钟计数器
09	外部硬件	0024h	键盘设备输入
0A	外部硬件	0028h	保留
0B	外部硬件	002Ch	串行口 2(COM2)
0C	外部硬件	0030h	串行口 1(COM1)
0D	外部硬件	0034h	硬盘中断
0E	外部硬件	0038h	软盘中断
0F	外部硬件	003Ch	打印机控制设备

表 11-1 硬件中断表

中断号	类别	偏移地址	目的描述
05	ROM-BIOS 软件	0014h	打印屏幕
10	ROM-BIOS 软件	0040h	视频 I/O
11	ROM-BIOS 软件	0044h	设备配置检查
12	ROM-BIOS 软件	0048h	内存大小检查
13	ROM-BIOS 软件	004Ch	硬盘/软盘驱动
14	ROM-BIOS 软件	0050h	通讯口 I/O 驱动
15	ROM-BIOS 软件	0054h	磁带 I/O(AT 辅助功能)
16	ROM-BIOS 软件	0058h	键盘 I/O 驱动
17	ROM-BIOS 软件	005Ch	打印机 I/O 驱动
18	ROM-BIOS 软件	0060h	ROM BASIC/非 IBM 机上不允许
9	ROM-BIOS 软件	0064h	装入加载
1A	ROM-BIOS 软件	0068h	TOD 时钟控制
1B	ROM-BIOS 软件	006Ch	Control-Break 处理程序
1C	ROM-BIOS 软件	0070h	时钟控制
1D	非中断向量	0074h	视频初始化表的地址
1E	非中断向量	0078h	磁盘参数表地址
1F	非中断向量	007Ch	ASCII 图形字符表 128-255

表 11-2 BIOS 软件中断表

中断号	类别	偏移地址	目的描述
20	软中断	0080h	程序终止(已过时)
21	软中断	0084h	DOS 功能调用 DOS 使用
22	软中断	0088h	终止向量/DOS 使用
23	软中断	008Ch	Control-C 处理程序/DOS 使用
24	软中断	0090h	致命错误处理程序/DOS 使用
25	软中断	0094h	绝对磁盘读/DOS 使用
26	软中断	0098h	绝对磁盘写/DOS 使用
27	软中断	009Ch	内存驻留(已过时)/DOS 使用
28 - 5F	软中断	00A0h - 0017Ch	DOS 保留/DOS 使用
60 - 67	软中断	010h - 019Ch	可使用的/用户可定义
68 - 7F	软中断	01A0h - 01FCh Stat	

表 11-3 DSO 软件中断表

表 11-1, 表 11-2, 表 11-3 实际上可看作是整张中断向量表的前半部分, 表中给出了中断的类别, 中断号以及相应中断在中断向量表中的偏移地址。

每次我们启动计算机, DOS 都将标准中断例程的入口地址填到中断向量表中。可是, 只要计算机一开始运行, 我们就可以修改中断向量表中的地址以指向自己写的中断处理过程, 然后中所将执行我们的过程而不是缺省的过程。简而言之, 只要我们修改中断向量表中的地址, 我们就能取代 DOS 和 BIOS 而承担起我们的计算机的基本功能。

在我们结束讨论中断之前, 还想特别提一下中断向量表的一种特殊用途。中断向量表原是用来存放中断处理例程的入口地址的, 但 IBM 却用它存放了三个根本不是程序而是数据项的地址。这就是说, 中断向量表成了存储重要地址的统一场所, 通常放的程序的地址, 但必要的时候, 也可存放数据地址。1D, 1E, 1F 等三个中断号相应的中断向量单元被预先空出来, 存放了三张重要数据表的起始地址, 参见表 11-2。如果某个程序请求这三个中断之一的話, 计算机就会跳转到某个数据表上去, 并试图将它作为一个程序来运行。

### § 11.3 软中断: DOS 单元与操作系统公共服务

软件中断是最有趣的。通常当某个程序需要用到另一个程序或子例程时, 它必须把计算机的控制交给那个程序。习惯上这叫做调用。为了调用某个例程, 我们的程序必须知道该例程的地址, 但被调例程不需要知道我们的地址, 因为调用机制会自动产生返回地址。为了转到某个例程序从那儿返回, 我们只需提供一张“单向”的入场券。软件中断的思想是要在两个方向上都能够自由进行, 既能行用某个例程并返回, 又使双方都无需知道对方的地址。

软件中断是靠一个程序有意识地产生一个中断来做到这一点的。如果我们的程序需要让

别的程序帮它计时的话，它们无需知道那个计时程序的地址，它们只需知道，该计时程序是 26(1Ah)-号中断调用的。

软中断来执行所有公共的服务，包括 IBM PC 的 ROM-BIOS 中固有的服务，以及由 DOS 操作系统提供的服务。

采用中断而不直接用地址的理由有两个。最重要的理由是，必要时可以变动中断所调用的程序，即改变它的规模和地址，而不必修改调用该例程的程序。另一个理由是为了能控制这些服务程序，如果我们在某个程序中希望修改一个服务的功能，我们就可以临时用一个新例程的地址来取代原例程的中断向量。

软件中断对个人计算机的操作非常重要。事实上，我们将化相当大的篇幅来描述软中断提供 BIOS 和 DOS 服务，为使这些服务能够与全为我们所用，本书将给出 Turbo Pascal 的支持程序和接口，并在 Turbo pascal 背景下介绍 BIOS 和 DOS 服务的功能。

### § 11.3.1 Turbo Pascal 的 DOS 单元

Turbo Pascal 6 提供了一个叫做 DOS 的标准单元，它包含了调用特定 DOS 和 BIOS 服务的例程以及我们自己调用服务所需要的数据结构和过程。利用这些例程，我们可以得到文件信息，目录列表，也可以为系统和个别文件设置时间和日期，还可以做其它许多工作。本节将描述这些新的过程以及说明我们如何使用它们。

DOS 单元中的某些过程需要特殊的数据类型用于定义变量。DOS 单元也包含两个过程，MsDos 和 Intr，我们能用它们调用特定的 DOS 和 BIOS 服务，这两个过程都使用类型为 Registers 的参数。有了 MsDos，Intr 和 Registers 数据结构，我们就可以利用所有 DOS 和 BIOS 提供的服务。另外，Borland 还为用得最普遍的服务提供了全套易于调用的过程使得访问那些 DOS 和 BIOS 服务更加容易。这些过程也已经封装在 DOS 单元里了。

#### § 11.3.1.1 DOS 单元的常量和类型

DOS 单元包含许多有助于简化程序设计的常量，这些常量分为三类：标志常量(用于解释 CPU 的标志寄存器)，文件方式常量(由 Turbo Pascal 的文件处理过程使用)以及文件属性常量(用于解释一个文件的属性字节)。这些常量的说明如下所示：

{ Flags Constants }

Const

```
FCarry      = $0000; (* Carry Flag      *)
FParity     = $0004; (* Parity Flag     *)
FAuxiliary  = $0010; (* Auxiliary Flag *)
FZero       = $0000; (* Zero Flag       *)
FSign       = $0080; (* Sign Flag       *)
FOverflow   = $0800; (* Overflow Flag  *)
```

{ File Mode Constants }

Const

```

fmClosed = $D7B0; (* File Closed *)
fmInput = $D7B1; (* File Opne for Input *)
fmOutput = $D7B2; (* File Open for Output *)
fmInOut = $D7B3; (* File Opne for Input and Output *)

```

{ File Attribute Constants }

Const

```

ReadOnly = $01;
Hidden = $02;
SysFile = $04;
VolumeID = $08;
Directory = $10;
Arhive = $20;
AnyFile = $20;
End;

```

DOS 单元包含了几个数据类型的说明，它们都是 DOS 单元中的例程所使用的。FileRec 数据类型用于类型和无类型文件变量，而 TextRec 数据类型则用于文本文件(text-file)变量。

Type

```

{ Typed and untyped }
FileRec = Record
Handle : Word;
Mode : Word;
RecSize : Word;
Private : Array [1..26] Of Byte;
End;

```

{ Textfile Record }

```

TextBuf = Array [0..127] Of Char;
TexRec = Record
Handle : Word;
Mode : Word;
Bufsize : Word;
Private : Word;
BufPos : Word;
BufEnd : Word;

```

```

BufPtr : ^ TextBf;
OpenFunc : Pointer;
InOutFunc : Pinter;
FlushFunc : Pointer;
CloseFunc : Pointer;
UserData : Array [1..16] Of Byte;
Name : Array [0..79] Of Char;
Buffer : TextBuf;
End;

```

FileRec 和 TextRec 两者均包含 Mode 域, 它可以用刚才描述的文件方式常量来解释。

Registers 数据类型用于 Intr 和 MsDos 例程调用 DOS 和 BIOS 服务:

Type

```

Registers = Record
Case Integer Of
0 : (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags : Word);
1 : (AL,AH,BL,BH,CL,CH,DL,DH : Word);

```

Registers 数据类型中的每个域都指明一个 CPU 寄存器, 通过设置该记录中的值, 我们就能象在汇编语言中一样调用系统级服务。

DOS 单元也包括用于时间和日期函数(Date Time)和目录操作(SearchRec)的数据类型。DateTime 数据类型在 GetTime 和 SetTime 过程中使用, 这两个过程读取和设置系统时钟的时间和日期。

The DateTime Type

Type

```

DateTime = Record

Year, Month, Day, Hour, Min, Sec : Integer;
End;

```

The SearchRec Type

Type

```

SearchRec = Record
Fill : Arra[1..21] oF bYTE;
Attr : Byte;
Time : LongInt;
Size : LongInt;
Name : String[12];

```

End;

SearchRec 记录类型在 FindFirst 和 FindNext 两个过程中使用,这两个过程读取目录中的文件项。SearchRec 记录的域包括文件属性(Attr),时间标记(Time),文件大小(Size)以及文件名(Name)。

DOS 单元还说明了三个 String 数据类型: DirStr, NameStr 和 ExStr。DirStr 类型用于存放文件名的目录路径部分(例如 C:/TP/TEMP);NameStr 用于文件名;ExStr 用于文件扩展名。

```
Type
    DirStr = String
        64;
    NameStr = String
        81;
    ExStr = String
        31;
```

这些类型用于 Fsplit 过程,该过程要求一个完整的文件指定字符串并返回分离的路径,文件名和扩展名。

#### § 11.3.1.2 DosError 变量

DOS 单元的许多例程用 DosError 报告错误。DosError 是一个 Integer(整型)变量, DosError 中存放的值是错误代码。参见表 11-4。

DOS 错误码	含义
0	无错
2	找不到文件
3	找不到路径
5	不允许存取
6	无效文件指针
8	没有足够的内存
10	环境无效
11	格式无效
18	没有其它文件

表 11-4 DosError 的可能取值

#### § 11.3.1.3 DOS 单元的过程和函数

DOS 单元实现了非常有用的操作系统和文件处理例程,这些都是对标准 Pascal 的扩充,本节我们分类简短地给出这些过程和函数,在本章中还将进一步介绍它们的具体使用实例。

##### 1. 日期和时间过程

有关日期和时间的过程列于表 11-5



过程	功能
GetDateKL	返回操作系统的当前日期
GetFTime	返回文件最后修改的日期与时间
GetTime	返回操作系统的当前时间
PackTime	将 DateTime 记录转换成一个可供 SetFTime 使用的 4 字节压缩的日期 和时间型长整数, DateTime 记录的域没有经过范围检查
SetDate	设置操作系统的当前日期。
SetFTime	设置文件的最后修改时间。
SetTime	设置操作系统的当前时间
UnpackTime	将由 GetFTime, FindFirst 或 FindNext 返回的 4 字节压缩的日期时间整数转换成非压缩的 DateTime 记录。

表 11-5 DOS 单元的日期时间过程

## 2. 中断支持过程

有关中断支持的过程列于表 11-6。

过程	功能
GetIntVec	返回存贮于指定中断向量中的地址
Intr	执行指定的软件中断
MsDos	执行一个 DOS 函数调用
SetIntVec	将指定中断向量设置成指定的地址。

表 11-6 DOS 单元的中断支持过程

## 3. 磁盘状态函数

有关磁盘状态的函数列于表 11-7 中

函数	功能
DiskFree	返回指定盘的空闲字节数
DiskSize	返回指定盘的总字节数

表 11-7 DOS 单元的磁盘状态函数

## 4. 文件处理过程和函数

有关文件处理的过程与函数列于表 11-8 和表 11-9 中。

过程	功能
----	----

FindFirst	在指定的(或当前的)目录中查找匹配给定主件名和属性的第一个文件
FindNext	返回匹配上次调用 Find First 指定的文件和属性的下一个文件。
FSplit	将文件名分成目录、文件名和扩展名三部分
GetAttr	返回文件的属性
SetAttr	设置文件的属性

表 11-8 DOS 单元的文件处理过程

函数	功能
FExpand	取一个文件名并返回全文件名(驱动器, 目录和扩展名)
FSearch	在目录表中查找文件

表 11-9 DOS 单元的文件处理函数

#### 5. 进程处理过程与函数

有关进程处理的过程与函数列于表 11-10 和表 11-11 中。

函数	功能
DosExitCode	返回子进程出口码

表 11-10 Dos 单元的进程处理函数

过程	功能
Exec	执行带特定命令行的特定程序
Keep	终止程序执行, 使其驻留内存
SwapVector	交换保存的中断向量和当前向量

表 11-11 DOS 单元的进程处理过程

#### 6. 环境处理函数

有关环境处理的函数列于表 11-12 中。

函数	功能
EnvCount	返回 DOS 环境变量的个数
EnvStr	返回特定环境变量
GetEnv	返回指定环境变量的值

表 11-12 DOS 单元的环境处理函数

## 7. 其它过程和函数

所有其它的过程和函数列于表 11-13 中。

过程或函数	功能
GetCBreak	返回 DOS 中 Control-Break 的检查状态
GetVerify	返回 DOS 验证位的状态
SetCBreak	设置 DOS 中 Control-Break 的检查状态
SetVerify	设置 DOS 验证位的状态
DosVersion	返回 DOS 版本号

表 11-13 DOS 单元的杂项过程的函数

### § 11.3.2 直接访问 BIOS 和 DOS 服务

Turbo Pascal 的 DOS 单元提供了两个过程, MsDos 和 Intr, 利用它们我们就能直接调用 BIOS 和 DOS 服务。这两个过程都必须带有一个 Registers 类型的参数。Intr 可以调用所有 BIOS 例程, 它的一般形式为:

Intr(<中断号>, <寄存器变量>);

其中<中断号>代表要执行的中断的类别, <寄存器变量>可以设置来说明要调用哪一个服务。MsDos 专门用来调用 DOS 服务, 它看上去很象 Intr 过程, 不过中断号参数联消了。MsDos 的一般形式为:

MsDos(<寄存器变量>);

因为几乎所有的 DOS 服务都通过中断 21h 取得, 因此不再需要中断号参数。下面的 Intr 命令与 MsDos 命令是等价的:

Intr(\$21, <寄存器变量>);

有时候, DOS 和 BIOS 服务之间的区别并不是很清晰, 例如, DOS 和 BIOS 都提供了在屏幕上显示信息, 从键盘接收信息以及在磁盘上存贮信息等服务。然而 DOS 服务一般处于较高的水平, 不依赖于具体硬件, 也就更加有用一些。

Registers 变量是发挥 DOS 和 BIOS 服务能力的关键, 这个变量包括了匹配多数 8086/8088 寄存器的域:

Registers 数据类型只包含在 BIOS 和 DOS 服务中用到的那些寄存器中。这个记录中有两个变体部分: 一个由表示整个寄存器的字变量组成, 另一个由定义通用寄存器的高、低部分的字节组成。例如, 两个 BYTE 变量 AL 和 AH 访问包含 AX 的同一片内存区域。由于 8088 微处理器以逆序存放字中的字节, 因此低位字节(AL)先于高位字节(AH), 如果整数 1 存放在 AX 中, 那么它内存中看上去就好象是这样的:

Type

Registers = Record

Case Integer Of

0: (AX, BX, CX, DX, BP, SI, DI, DS, ES, Flags ; Word);

1; (AL,AH,BL,BH,CL,CH,DL,DH ; Byte);

End;

在我们能调用 MSDOS 或 Intr 之前,我们必须设置寄存器组变量以规定我们想要哪一个服务和我们想如何执行这个服务。例如,为了选择一个 DOS 服务,要把该服务的代码放入 AH 寄存器。这里给出一个设置系统日期的 DOS 服务 2Bh 的使用实例:

Program SetTime;

Uses DOS, TRT;

Var

Regs ; Registers;

Begin

ClrScr;

FillChar(Regs,SizeOf(Regs),0);

With Regs Do

Begin

AH := \$2B;

DH := 12; (\* Month \*)

DL := 31; (\* Date \*)

CX := 1991; (\* Year \*)

End;

MsDos(Regs); (\* Call the DOS service \*)

If Regs.AL <> 0 Then

WriteLn('Error! ');

Else

WriteLn('Date has been set. /');

WriteLn;

Write('Press ENTER... ');

ReadLn;

End.

过程首先用语句

Fill Char(Regs, Sizeof(Regs),0)将寄存器组初始化为全 0,然后把设置系统日期的代码 2Bh 送入 AH,接着把日期信息放到 DH, DL 和 CX 中。

MsDos 接受寄存器组变量作为参数并调用 DOS 过程将系统日期更新为 1991 年 12 月 31 日。如果 DOS 服务检测到一个错误,则它返回在 AL 中放有错误码的寄存器组变量。语句

If Regs.AL <> 0 Then

WriteLn('Error! ');

检查这个代码, 如果它不等于 0 则表明出现了错误。

MsDos 过程用于 DOS 服务, 而 Intr 过程则是用于 BIOS 服务。Intr 接受两个变量: 中断号和寄存器组变量, 例如, 通过设置寄存器 AH 为 5 并调用中断 5, 象下面这样:

```
Fill Char(Regs, Sizeof(Regs), 0);
Regs.AH := 5;
Intr(5, Regs);
```

就可以调用打印屏幕内容的 BIOS 服务。同样的服务也可以通过同时按下 SHIFT 和 PRTSC 键来调用。在这个服务中, 没有错误指示在寄存器组中返回。

使用 DOS 和 BIOS 服务大大增强了我们用 Turbo Pascal 处理事务的能力, 但学习使用它们也要相应地占用一些时间。本书的其余部分将致力于说明如何使用最有用的 DOS 和 BIOS 服务。

### § 11.3.2.1 磁盘驱动器服务

磁盘操作系统(DOS)的主要目的是管理我们的计算机的磁盘驱动器和文件。可庆幸的是, Turbo Pascal 的标准过程已处理了最困难的磁盘相关任务, 象读文件和写文件。本节给出几个 Turbo Pascal 未提供但能改进我们的程序的 DOS 功能。

#### 1. 报告磁盘的空闲可用空间

DOS 服务 36h 指明磁盘上还有多少可用空间。DL 寄存器选择要检查的磁盘, 0 指明缺省盘, 1 指明驱动器 A, 2 指明驱动器 B, 依次类推。在 MsDos 调用之后, 通用寄存器中含有以下信息:

AX: 每个分配簇的扇区数  
BX: 未使用的簇数  
CX: 每扇区字节数  
DX: 总簇数

使用这些值, 我们可以很容易地计算出空闲磁盘空间的总量等于

$\text{LongInt}(\text{AX}) * \text{BX} * \text{CX}$ 。在等式中的 LongInt 类型强制是为了避免整数溢出。如果指定了一个无效驱动器, Turbo Pascal 在 AX 中返回 \$FFFF。

这里给出的 FreeDiskSpace 函数可以报告一个驱动器的空闲字节量:

```
Program DiskSpace;
Uses CRT, DOS;
Var
  Drive : Char;

(* *)

Function FreeDiskSpace(Drive : Char) : LongInt;
Var
  Regs : Registers;
Begin
  FillChar(Regs, SizeOf(Regs), 0);
```

```

With Regs Do
  Begin
    AH := $36;
    DL := Ord(UpCase(Drive)) - 64;
  End;

MsDos(Regs);

With Regs Do
  If AX = $FFFFFF Then
    FreeDiskSpace := -1
  Else
    FreeDiskSpace := LongInt(AX) * BX * CX;
  End;

( * * * * * )

Begin
  ClrScr;
  Write( 'Which Drive? (A/B/C/D): ');

  ReadLn(Drive);
  Drive := UpCase(Drive);
  WriteLn(FreeDiskSpace(Drive); 0 bytes free. ');
  WriteLn;
  Write( 'Press ENTER.. ');
  ReadLn;
End.

```

前一个过程的参数是指明要检查的驱动器的一个字符，为了将驱动器号赋给 DL，该过程先将参数 Drive 改为大写字母，再转换为 ASCII 值，然后减去 64。如果 drive 等于 a，则被改为 A，它的 ASCII 值是 65，从 65 中减去 64 等于 1，这是 DL 要求的正确的驱动器号。

MsDos 调用之后，该过程检查 AX 寄存器，如果 AX 中是 \$FFFF，表明该过程期间出现了一个错误，函数返回值为 -1，如果无错误，则函数计算空闲磁盘空间总量。

## 2. 取得和设置文件属性

磁盘文件可以具有 DOS 提供的六个属性之一，它们是：只读，隐藏，系统，卷标，子目录和归档。文件属性包含在一个字节内，不同的属性由不同的位控制。

第 1 位表示文件的只读状态，只读文件不能用任何方式改变，DOS 阻止任何覆盖(写)或删除只读文件的企图，恰似一个写保护标签保护软盘。

第 2 位告诉我们一个文件是否是隐藏的。隐藏文件通常都含有一些敏感信息，它们会被 DOS 忽略：既不能被 Dir 命令列出，也不能被显示或删除。结果是，除非用户有一个程序能找到它们，否则隐藏文件是看不见的。虽然 DOS 不承认隐藏文件，但 Turbo Pascal 允许它们用于输入或输出。

第 3 位控制系统属性。系统文件象隐藏文件一样不被 DOS 命令所承认。但系统属性没有实际的作用，它仅仅是从 CP/M 中转过来的。

第 1 位与卷标有关，卷标是标识软盘或硬盘的语句，它是用户格式化磁盘时的可选设置。

属性字节的第 5 位指示文件是一个子目录，子目录文件不是存放数据的，而是 DOS 用来保存目录与子目录的踪迹。

第 6 位，归档属性位，在文件最初建立时置位，当文件已经用 DOS 命令 Backup 做了付本时该位关闭并保持到文件内容被修改。归档位允许我们只为上次备份以来有所修改的文件做备份。

第 7 位和第 8 位不用。

为了找出一个文件有什么样的属性，或设置成我们想要的属性，用 DOS 服务 43h。AL 寄存器的值控制着要执行的动作：0 是报告文件属性，1 是设置文件属性。

报告文件属性时，43h 服务将属性字节返回在 CL 寄存器内，通过测试各个位，我们就可以判定每个属性的状态。设置文件属性时，我们必须建立一个属性字节，并在调用 MsDos 之前把它放到 CL 寄存器中。

无论设置还是报告一个文件的属性字节，都必须将包含文件名的 ASCIIZ 字符串的段地址和偏移量装入寄存器组 DS:DX 中。一个 ASCIIZ 串是一个由二进制 0 结束的字母数组。我们可以将一个 Turbo Pascal 字符串加上 #0 作为一个 ASCIIZ 串，但 Turbo Pascal 串有一个长度字节，而 ASCIIZ 串没有，因此，我们必须使用串中第一个字符的地址作为 ASCIIZ 串的起点。

在下面的过程中，DOS 服务 43h 设置了六个布尔参数，每个可能的文件属性用一个：

```
Program FileAttributes;
```

```
Uses CRT, DOS;
```

```
Var
```

```
  Fname : String;
```

```
  RO, Hidden,
```

```
  Sys, Vol,
```

```
  SubDir, Arch, Error : Boolean;
```

```
( * * * * * )
```

```
Procedure GetFileAttributes(FileName : String;
```

```
  Var RO,
```

```
  Hidden,
```

```
  Sys,
```

```

Vol,
SubDir,
Arch,
Error : Boolean);

```

```

Var

```

```

    Regs : Registers;

```

```

Begin

```

```

    FillChar(Regs,SizeOf(Regs),0);

```

```

    FileName := FileName + #0;

```

```

    With Regs Do

```

```

        Begin

```

```

            AH := $43;

```

```

            DS := Seg(FileName);

```

```

            DX := OfS(FileName) + 1;

```

```

        End;

```

```

    MsDos(Regs);

```

```

    Error := (Regs.AL in [2,3,5]);

```

```

    RO := (Regs.CL And $01) > 0;

```

```

    Hidden := (Regs.CL And $02) > 0;

```

```

    Sys := (Regs.CL And $04) > 0;

```

```

    Vol := (Regs.CI And $08) > 0;

```

```

    SubDir := (Regs.CI And $10) > 0;

```

```

    Arch := (Regs.CI And $20) > 0;

```

```

    End;

```

```

    ( * * * * * )

```

```

Begin

```

```

    CluSer;

```

```

    Write( 'Which file? ');

```

```

    ReadLn(Fname);

```

```

    GetFileAttributes(Fname,

```

```

        RO,

```

```

        Hidden,

```

```

        Sys,

```



```

        Vol,
        Arch,
        Error);

If Error Then
    WriteLn( 'Error! ');
Else
    Begin
        WriteLn(Fname, ' has these attributes: ');
        WriteLn( 'Read only:      ',RO);
        WriteLn( 'Hidden:        ',Hidden);
        WriteLn( 'System file:    ',Sys);
        WriteLn( 'Volume label:   ',Vol);
        WriteLn( 'Subdirectory:   ',SubDir);
        WriteLn( 'Archive:       ',Arch);
    End;

WriteLn;
Write( ' Press ENTER... ');
ReadLn;
End.

```

DOS 服务 43h 报告的错误条件如下:

当文件未找到时, AL 寄存器为 2;

当路径未找到时, AL 寄存器为 3;

当文件不允许存取时, AL 为 4。

下面的过程设置一个文件的属性字节。过程为四个文件属性提供了四个布尔参数, 它不包括卷标和子目录属性, 因为它们不能由这个 DOS 服务设置。

```

Program FileAttributes;
Uses CRT, DOS;
Var
    ch : Char;
    Fname : String;
    RO, Hidden,
    Sys, Arch : Boolean;

( * * * * * )

Procedure SetFileAttributes(FileName : String;

```

```

                                Var RO,
                                Hidden,
                                Sys,
                                Arch : Boolean);

Var
    Regs : Registers;

Begin
    FillChar(Regs,SizeOf(Regs),0);
    FileName := FileName + #0;

    With Regs Do
        Begin
            AH := $43;
            AL := 1;
            DS := Seg(FileName);
            DX := OfS(FileName) + 1;

            If RO Then
                CL := (CL Or $01);
            If Hidden Then
                CL := (CL Or $02);
            If Sys Then
                CL := (CL Or $04);
            If Arch Then
                CL := (CL Or $20);
            End;

        MsDos(Regs);

    End;
    ( * * * * * )
    Begin
        ChrScr;
        Write( 'Which file?: ');
        ReadLn(Fname);

        Write( 'Set to read only? (Y/N) ');
        ReadLn(ch);
        RO := UpCase(ch) = 'Y';

```

```

Write( 'Set to hidden (Y/N) ');
ReadLn(ch);
Arch := UpCase(ch) = 'Y';
Write( 'Set to system file(Y/N) ');
ReadLn(ch);
Sys := UpCase(ch) = 'Y';
SetFileAttributes(Fname,
                    Ro,
                    Hidden,
                    Sys,
                    Arch);
WriteLn(Fname, ' has been set to these attributes: ');
WriteLn( 'Read only: ',RO);
WriteLn(Hidden: ' ',Hidden);
WriteLn(System file: ',Sys);
WriteLn(Archive: ' ',Arch);
WriteLn;
WriteLn( 'Press ENTER... ');
ReadLn;
;End.

```

如果我们想让自己的文件保密,或都想显露已经隐藏的文件,或者想设定只读状态,那么改变文件的属性是很有用的。

### 3> 目录列表

显示一个磁盘目录需要三个不同的 DOS 服务,还需要理解程序段前缀(PSP)以及磁盘传送区(DTA),DTA 是 PSP 的一部分。

每次 DOS 启动一个程序时,都把分配给程序的前 256 字节内存作为程序的 PSP。由于 PSP 包含高度技术性的信息,因此通常除了 DTA 部分之外程序员从不接触它们。DTA 是一个 128 字节的缺省缓冲区用于某些 DOS 操作,如读一个磁盘目录,这时 DTA 中的信息如表 11-14 所示。

描述	偏移量	长度
DOS 指用数据	0	21
文件属性	21	1
文件的时间戳	22	2
文件的期戳	24	2
以字节表示的文件大小	26	4
文件名和扩展名	30	13

表 11-14 目录列表时的 DTA 内容

文件名及其扩展名构成了 DTA 的最后一个域，文件属性，时间，日期和文件大小能被读并翻译成更为完整的目录列表。

在从 DTA 中得到信息之前，我们必须知道它的地址，这是由 DOS 服务 2Fh 报告的。2Fh 服务执行时把 DTA 的段地址放入 ES 中，把 DTA 的偏移量放入 BX，如下所示：

```
Regs. AH := $ 2F;
MsDos(Regs);
DTAseg := Regs. ES;
DTAofs := Regs. BX;
```

以上程序段将 DTA 段地址存贮在变量 DTAseg 中，而将偏移量存贮在 DTAofs 变量中。这些变量与 Turbo Pascal 的标准数组 Mem 一起使用可以从 DTA 中获取信息，例如：

Mem[DTA Seg:DTA ofs+21]指向 DTA 中文件属性字节的单元。

DOS 服务 4Eh 查找目录中第一个匹配的文件并用该文件的信息填写 DTA，可是在调用 4Dh 前，寄存器 DS 和 DX 中必须包含路径和文件名所在 ASCII 串的段地址和偏移量。如下面代码段所示：

```
Mask-In := Mask-In + #0;
With Regs Do
  Begin
    AH := $ 4E;
    DS := Seg(Mask-In);
    DX := Ofc(Mask-In+1);
    CL := $ 00;
  End;
```

```
MsSDos(Regs);
IF Regs. AL <> 0 Then Exit;
```

CL 寄存器告诉 DOS 应该包括在本次查找中的文件类型，如果该寄存器为 0，DOS 只查找标准文件。为了在目录表中包括隐藏或系统文件，CL 寄存器应按表 11-15 中指导的值设置。

如果 CL 设置成 16h (2h, 4h 与 10h 之和)，则目录列表中应包括隐藏文件，系统文件和子目录。如果 AL 寄存器返回一个非零值，那么在目录中没有找到与文件说明串匹配的文件项。

要包括的属性	CL 的值
隐藏	\$ 02
系统	\$ 04
卷标	\$ 08
子目录	\$ 10

表 11-15 CL 寄存器的 DOS 文件属性设置

程序 Directory 使用 DirList 过程以便为用户输入的文件说明建立一个文件名数组：

Program Directory;

Uses CRT, DOS;

Type

Dir—Files = Array [1..200] Of String[13];

Var

FileSpec : String;

i;

fc : Integer;

df : Dir—Files;

( \* \* \* \* \* )

Procedure DirList(Mask—In : String;

Var Name—List : Dir—Files;

Var File—Counter : Integer);

Var

i : Byte;

Regs : Registers;

DTAseg,

DTAofs : Word;

FileName : String[20];

Begin

FillChar(Regs,SizeOf(Regs),0);

File—Counter := 0);

Regs.AH := \$2F;

MdDos(Regs);

With Regs Do

Begin

DtAseg := ES;

DTAofs := BX;

End;

FillChar(Regs,SizeOf(Regs),0);

Mask—In := Mask—In + #0;

With Regs Do

Begin

```

    AH := $4E;
    DS := Seg(Mask-In);
    DX := Pfs(Mask-In) + 1;
    CL := $00;
    End;
MsDos(Regs);

If Regs.AL <> 0 Then Exit;

i := 1;
Repeat
    FileName[i] := Chr(Mem[DTAseg:DTAofs+29+i]);
    i := i + 1;
Until (FileName[i-1] < #32) Or (i > 12);

FileName[0] := Chr(i-1);
FileCounter := 1;
NameList[FileCounter] := FileName;

Repeat
    FileChar(Regs,SizeOf(Regs),0);
    With Regs Do
        Begin
            AH := $4F;
            CL := $00;
            End;
    MsDos(Regs);
    If Regs.AL = 0 Then
        Begin
            i := 1;
            Repeat
                FileName[i] := Chr(Mem[DTAseg:DTAofs+29+i]);
                i := i + 1;
            Until (FileName[i-1] < #32) Or (i > 12);

            Inc(FileCounter,1);
            FileName[0] := Chr(i-1);
            NameList[FileCounter] := FileName;
            End;

```

```

End;

( * * * * * )

Begin
  ClrScr;
  Repeat
    Write( 'Enter file spec: ');
    ReadLn(FileSpec);
    If FileSpec <> '' Then
      Begin
        DirList(FileSpec,df,fc);
        For i := 1 To fc Do
          WriteLn(df[i]);
          WriteLn;
        End;
      Until FileSpec = '';
  End.

```

过程 DirList 接受下列三个参数: mask—in, name—list 和 file—counter。字符串参数 mask—in 包含了要匹配的文件说明,例如 test.pas, \*.pas 或者???.pas 等。匹配文件说明的文件名存于 name—list 中,这是一个字符数组。整型量 file—counter 返回 DirList 查找中匹配的文件名的数量。

注意 DOS 服务 4Eh 仅定位于第一个文件, 4Fh 寻找所有后续的文件并继续定位,直到 AL 寄存器出现非零值表明本目录中不再有匹配的文件为止。

### § 11.3.2.2 视频服务

多数用户判断一个程序的好坏几乎整个依据它所采用的视频特性,因为设计良好且有吸引力的显示使程序更易于使用。不幸 Turbo pascal 只提供有限的屏幕控制能力。本节给出的过程扩展了我们对监视器的控制并使我们有可能制作出更复杂的视频显示。

#### 1. 报告当前视频模式

屏幕控制的一个基本的方面是判定计算机具有的视频适配器的类型,主要有单色适配器、彩色图形适配器以及后来的 PCjr 和增强彩色图形适配器。

BIOS 中断 10h 报告正在用的视频适配器类型,下面给出的函数 CurrentVidMode 就是使 Int 10h 的一个例子。

```

Program VideoMode;
Uses CRT, DOS;

( * * * * * )

```

```

Function CurrentVidMode : Char;
Var
    Regs : Registers;
Begin
    FillChar(Regs,SizeOf(Regs),0);
    Regs.AH := $0F;
    Intr($10,Regs);

    Case Regs.AL of
        1..6 : CurrentVidMode := 'C'; (* CGA *)

        7 : CurrentVidMode := 'M'; (* Monochrome *)
        8..10 : CurrentVidMode := 'P'; (* PCjr *)
        13..16 : CurrentVidMode := 'E'; (* EGA *)
    End;

End;

(******)
Begin
    ClrScr
    WriteLn('Current video mode is: ',CurrentVidMode);
    Writeln;
    Write('Press ENTER... ');
    ReadLn;
End.

```

在中断调用之前，函数将 AH 寄存器设置成 0Fh，中断将屏幕宽度(即每行字符数)放在 AH 中，视频方式放在 AL 中，视频页号放在 BH 中。过程通过检查 AL 来判定视频方式，如果这个寄存器等于7，屏幕是单色显示器，并且函数返回字母 M。值1 到6 指示彩色图形监视器(C)，8 到10 意味着 Pcjr(P)，而13 到16 则是增强图形(E)。

当你开始直接向视频存储区写信息时，了解显示类型是基本的前提。

## 2. 设置光标大小

在程序运行的某一时刻，最好不要显示光标，而在另一时刻，大光标比小光标更合理些。典型地，一个光标由两条扫描线组成，可是在彩色图形适配器能够用8 条扫描线来显示光标，而单色适配器可以多至14条。扫描线用得越多，光标越大；不用任何扫描线，光标就消失了。

为了设置光标大小，使用 BIOS 中断10h 并把 AH 置成1，在寄存器 CH 中放上起始扫描线序号且在寄存器 CL 中放上结束扫描线的序号。彩色图形适配器使用8 条扫描线(0到7)，单色适配器使用14 条线(0到13)，较低(序号)的扫描线出现在靠近屏幕顶端，例如，彩色图形监视器上的小光标由扫描线6 和7 组成，用了底部的两条扫描线。



```

Program Cursor;
Uses DOS, CRT;

```

```

( * * * * * )

```

```

Procedure CursorSize(Stype, Size : Char);

```

```

Var

```

```

    Reg : Registers;

```

```

    i : Integer;

```

```

Begin

```

```

    Size := UpCase(Size);

```

```

    If UpCase(Stype) = 'M' Then

```

```

        i := 6

```

```

    Else

```

```

        i := 0;

```

```

    Regs.AH := $01;

```

```

    Case Size Of

```

```

        'O' :

```

```

            Begin

```

```

                Regs.CH := $20;

```

```

                Regs.CL := $7+i;

```

```

            End;

```

```

        'S' :

```

```

            Begin

```

```

                Regs.CH := $6+i;

```

```

                Regs.Cl := $7+i;

```

```

            End;

```

```

    End;

```

```

Intr($10, Regs);

```

```

End;

```

```

( * * * * * )

```

```

Begin

```

```

    ClrScr;

```

```

    WriteLn('Big cursor');

```

```

    CursorSize('C', 'B');

```

```

    WriteLn;

```

```

WriteLn;
Write( 'Press ENTER...' );
ReadLn;

WriteLn;
writeLn;
WriteLn( 'No cursor' );
CursorSize( 'C', 'O' );
WriteLn;
Write( 'Press INTER...' );
WriteLn;
Write( 'Press ENTER...' );
ReadLn;

WriteLn;
WriteLn;
WriteLn( 'Small cursor' );
CursorSize( 'C', 'S' );
WriteLn;
Write( 'Press ENTER...' );
ReadLn;

End.

```

这个过程按照你传递给它的参数来设置光标的大小，参数 *Stype* 可以等于 *M* 以指明单色显示器，可以等于 *C* 以指明彩色图形显示器。参数 *Size* 能取三种值：*B* 表示大，*S* 表示小，或0表示无。

如果计算机用了一个单色适配器，变量 *i* 设置为6，其它情形设置为0。如果不要光标，只须简单地将 *CH* 和 *CL* 都设置成20h，而大光标则通过将 *CH* 置成0，将 *CL* 置成7(彩色图形适配器)或13(单色适配器)来建立。如果是要小光标，则对于彩色图形适配器将 *CH* 和 *CL* 置成6和7，对于单色适配器置成13和13。

### 3. 从屏幕上读字符

你可以用 BIOS 中断10h 从视频屏幕上读字符，下面的 *Screen Char* 函数说明如何用中断10h 从屏幕上读字符。

```

Program ScreenTest;
Uses DOS, CRT;
Var
    s : String;
    i : Integer;

```

( \* \* \* \* \* )

```
Function ScreenChar : Char;  
Var  
    Regs : Registers;  
Begin  
    FillChar(Regs,SizeOf(Regs),0);  
    Regs.AH := 8;  
    Regs.BH := 0;  
    Intr($10,Regs);  
    ScreenChar := Chr(Regs.AL);  
End;
```

( \* \* \* \* \* )

```
Begin  
ClrSer;  
  
WriteLn('ABCDE');  
  
s := '';  
For i := 1 To 5 Do  
    Begin  
        GotoXY(i,1);  
        s := s + ScreenChar;  
    End;  
WriteLn;  
WriteLn(S);  
  
WriteLn;  
Writ('Press ENTER...');  
ReadLn;  
End.
```

因为 Screen Char 读当前光标位置上的字符，指以你在调用中断之前必须正确地设置光标位置。将8置入AH，将0置入BH。寄存器BH中的数选择所用的视频页，但你最可能使用的是视频页0。

中断调用这后，光标位置上的字符的ASCII码返回在寄存器AL中。Screen Char 将ASCII码转换为字符并将其作为函数的返回结果。

### § 11.3.2.3 时间和日期功能

DOS 维护着一个保持时间和日期轨迹的内部时钟。当一个文件被创建或修改时，DOS 使用时钟以便在文件上打上日期和时间的戳记。表11-16中给出的 DOS 服务你能控制系统的日期和时间。

DOS 服务	功能
1Ah	报告系统日期
2Bh	设置系统日期
2Ch	报告系统时间
2Dh	设置系统时间

表11-16 DOS 系统时间和日期服务

在调用这些服务中的某个服务之前，必须在 AH 寄存器中指定合适的 DOS 服务码。

#### 1. 获得系统日期

报告当前系统日期的 DOS 服务2Ah，将系统日期信息放在表11-17所示的寄存器里。

寄存器	信息
AL	星期 (0=星期日)
CX	年
DH	月
DL	日

表11-17 DOS 服务2Ah 之后的寄存器内容

在下面给出的过程 Get System Date 中使用 DOS 服务2SAh 报告系统日期，然后将日期格式化为字符串形式，最后将日期返回给程序。

```
Program Date;
```

```
Uses CRT, DOS;
```

```
Var
```

```
  s : String;
```

```
( * * * * * )
```

```
Procedure GetSystemDate(Var date : String);
```

```
Var
```

```
  REgs : Registers;
```

```
  st, st2, st3, st4 : String[10];
```

```
Begin
```

```

FillChar(Regs,SizeOf(Regs),0);
Regs.AH := $2A;
MsDos(Regs);
With Regs Do
begin
    Case AL Of
        0 : st1 := 'Sunday';
        1 : st1 := 'Monday';
        2 : st1 := 'Tuesday';
        3 : st1 := 'Wednesday';
        4 : st1 := 'Thursday';

        5 : st1 := 'Friday';
        6 : st1 := 'Saturday';
    End;
    Str(CX, st2); (* Year *)
    Str(DH, st3); (* Month *)
    Str(DL, st4); (* Date *)
    End;
If emgtj(st3) = 1 Then
    st3 := '0' + st3;
    If Length(st4) = 1 Then
        st4 := '0' + st4;
    date := st1 + '-' + st3 + '-' + st4; End

    ( * * * * * )

Begin
ClrScr;

GetSystemDate(s);
WriteLn('The date is ',s);

WriteLn;

Write('Press ENTER.. ');ReadLn;

End;

```

GetSystemDate 使用了 Case 语句来判定合适的星期日期，年、月、日被转换成字符串，如

果日或月由单个数码组成的话，字符串中填上一个先导的零。

## 2. 设置系统日期

DOS 服务 2Bh 设置系统日期。在实施 MsDos 调用之前，我们必须将月信息插入到寄存器 DH 中，日信息插入到 DL 中，年信息插入到 CX 中。下面的过程给出 Turbo Pascal 如何使用这个服务。

```
Program Date;
Uses CRT, DOS;
Var
    Error : Boolean;

(* *)

Procedure SetSystemDate(Month, Day, Year : Integer;
                        Var Error : Boolean);
Var
    Regs : Registers;
Begin
    FillChar(Regs, SizeOf(Regs), 0);
    With Regs Do
        Begin
            AH := $sB;
            DH := Month;
            DL := Day;
            CX := Year;
        End;
    MsDos(Regs);
    Error := Regs.AL <> 0;

    (* *)

Begin
    ClrScr;

    SetSystemDate(1, 1, 1990, Error);

    If Error Then

        WriteLn('Error!')
    Else
```

```

    WriteLn( 'Date has been set. ');

WriteLn;
Write( 'Press ENTER... ');
ReadLn;

End.

```

如果我们输入了一个非法日期，就会发生错误，在这种情形下寄存器 AL 中返回一错误码，如果在 AL 中发现一个非零值，那么布尔参数 Error 设置为 TRUE(真)。

### 3. 获得和设置系统时间

DOS 服务 2Ch 报告系统时间。服务 2Dh 设置系统时间。报告和设置系统时间与报告和设置系统日期的操作非常相似。下面两个过程说明如何报告和设置系统时间。

```

Program Systime;
Uses DOS, CRT;
Var
    Hour,
    Minute,
    Second : Byte;
    Error : Boolean;
    s : String;

Procedure GetSystemTime(Var Time : String);

Var
    Regs : Registers;
    h, m, s : Word;
    st1, st2, st3, st4 : String[10];
Begin
    FillChar(Regs, SizeOf(Regs), 0);
    Regs.AH := $2C;
    MsDos(Regs);
    With Regs Do
        Begin
            Str(CH, st1);
            Str(CL, st2);
            Str(DH, st3);
            Str(DL, st4);

```

```

    End;

    If Length(st1) = 1 Then
        st1 := '0' + st1;

    If Length(st2) = 1 Then
        st2 := '0' + st2;

    If Length(st3) = 1 Then
        st3 := '0' + st3;

    If Length(st4) = 1 Then
        st4 := '0' + st4;

    Time := st1 + ':' + st2 + ':' + st3 + ':' + st4;
    End;

    ( * * * * * )

```

```

Procedure SetSystemTime(Hour, Minute, Second : Byte;
                        Var Error : Boolean);

```

```

Var
    Regs : registers;
Begin
    FillChar(Regs, SizeOf(Regs), 0);

```

```

    With Regs Do
        Begin
            AH := $sD;
            CH := Hour;
            CL := Minute;
            DH := Second;
        End;
        MSdos(Regs);
        Error := Regs.AL <> 0;
    End;

```

```

    ( * * * * * )

```

```

Begin

```



```

ClrScr;
Write('Hour : ');
ReadLn(Hour);
Write('Minute: ');
ReadLn(Minute);
Write('Secound: ');
ReadLn(Second)

SetSystemTime(Hour, Minute, Secound, Error);

GetSystemTime(s);
Write(' Press ENTER... ');
ReadLn;

End.

```

如果在设置时间时发生错误，寄存器 AL 中返回错误码，返回在 AL 中的任何非零码都指明一个错误条件。

#### 4. 获得和设置一个文件的时间和日期

DOS 服务 3Dh 能够报告或设置一个文件的时间和日期戳记。服务于磁盘文件的时间和日期功能是复杂的，因为必须使用文件句柄(file handle)，并且日期和时间被编码到一个数值中。文件句柄是用于处理磁盘输入和输出的 DOS 约定。

为了得到文件句柄，需要使用 DOS 服务 3Dh，它打开一个文件并返回文件句柄放在寄存器 AX 中。在下面程序中使用的函数 GetFileHandle 接受一个文件名并返回一个文件句柄：

```

Function GetFileHandle (FileName : String;
                        Var Error : Boolean) : Integer;
Var
    Regs : Registers;
    i : Integer;
Begin
    FileName := FileName + #0;
    FillChar(Regs, SizeOf(Regs), 0);
    With Regs Do
        Begin
            AH := $3D;
            AL := $00;
            DS := Seg(FileName);

```

```

    DX := OfS(FileName)+1;
End;

MsDos(Regs);

i := Regs.AX;

If (Lo(Regs.Flags) And $01) > 0 Then
    Begin
        Error := True;
        GetFileHandle := 0;
        Exit;
    End;
GetFileHandle := i;
End;

```

如果发生错误，GetFileHandle 返回零并把错误参数设置为 TURE(真)。如果没有发生错误，那么文件被打开，而且我们可以进一步报告或设置文件的时间和日期。可是在我们完成之前，我们必须确保用提供的文件句柄通过 DOS 服务 3Eh 关闭已被打开的文件。如下面过程所示：

```

Procedure CloseFileHandle(i : integer);
Var
    Regs : Registers;
Begin
    With Regs Do
        Begin
            AH := $3E;
            BX := i;
        End;
    MsDos(Regs);
End;

```

简言之，报告或设置文件的时间和日期是一个三步的过程：

- 1) 指开一个文件并存放好文件句柄。
- 2) 使用文件句柄报告或设置文件的时间和日期。
- 3) 关闭文件。

下面给出的两个过程 GetFileTime AndDa 和 SetFileTimeAnddate 说明如何使用 DOS 服务 57h。如果 AL 寄存器置成 0，TurboPascal 报告时间和日期；如果置成 1，TurboPascal 设置时间

和日期。在两种情形中，寄存器 BX 中都存放着文件句柄：

```
Program FileStamp;
Uses DOS, CRT;
Var
    Fname,
    Time—st,
    Day—st : String;
    Month, Day,
    Minute, Second : Word;
    Error : Boolean;

    ( * * * * * )

Function GetFileHandle(FileNmae : String;
                        Var Error : Boolean) : Integer;
Var
    Regs : Registers;
    i : Integer;
Begin
    FileName := FileName + #0;
    FillChar(Regs, SizeOf(Regs), 0);
    with Regs Do
        Begin
            AH := $3D;
            AL := $0;
            DS := Seg(FileName);
            DX := Ofs(FileName) + 1;
        End;

    MsDos(Regs);

    i := Regs.AX;

    If (Lo(regs.Flags) And $01) > 0 Then
        Begin
            Error := True;
            GetFileHandle := 0;
            Exit;
        End;
```

GetFileHandle := i;

End;

( \* \* \* \* \* )

Procedure CloseFileHandle(i : Integer);

Var

Regs : Registers;

Begin

With Regs Do

Begin

AH := \$3E;

BX := i;

End;

MSDOS(Regs);

End;

( \* \* \* \* \* )

Procedure GetFileTimeAndDate(File—Name : String;

Var Time—st,

Day—st : String;

Var Error : Boolean);

Var

Regs : Registers;

i : Integer;

st1,st2,st3 : String[4];

y,m,d,r,h,s,Time,Day : Word;

Begin

Error := False;

Time—st := '';

Day—st := '';

i := GetFileHandle(File—Name,Error);

If Error Then Exit;

With Regs Do

Begin

AH := \$57;

```

    AL := $00;
    BX := i;
    End;

MsDos(Regs);
CloseFileHandle(i);

( * Convert Time * )
r := Regs.CX;
h := r Div 2048;
r := r - (h * 2048);
s := r * 2;

Str(h:0,st1);
Str(m:0,st2);
If Length(st1) = 1 Then st1 := '0'+st1;
If Length(st2) = 1 Then st2 := '0'+st2;
If Length(st3) = 1 Then st3 := '0'+st3;
Time—st := st1+' '+st2+' '+st3;

) * Convert Day * )
r := Regs.DX;
y := (r Div 512) + 1980;
r := r - (y - 1980) * 512;
m := r Div 32;
r := r - (m * 32);
d := r;
Str(y:0,st1);
Str(m:0,st2);
Str(D:0,st3);
If Length(st1) = 1 Then st1 := '0'+st1;
If Length(st2) = 1 Then st2 := '0'+st2;
If Length(st3) = 1 Then st3 := '0'+st3;
Day—st := st2+'-' + st3+' '+st1;
End;

( * * * * * )

Procedure SetFileTimeAndDate(File—Name ; String;
                               Month, Day,

```

```

Year, Hour,
Minute, Second : Word;
Var Error : Boolean);

```

```

Var

```

```

  Regs : Registers;
  a,j,k : Word;
  t,d : Wprd;

```

```

Begin

```

```

Error := False;

```

```

i := GetFileHandle(File—Name,Error);

```

```

If Error Then Exit;

```

```

t := (Hour * 2048) + (Minute * 32) + (Second Div 2);

```

```

D := (Year—1980) * 512 + (Month * 32) + Day;

```

```

Wotj Regs Do

```

```

  Begin

```

```

    Ah := $57;

```

```

    AL := $01;

```

```

    BX := i;

```

```

    CX := t;

```

```

    DX := d;

```

```

  End;

```

```

MsDos(Regs);

```

```

CloseFileHandle(i);

```

```

End;

```

```

( * * * * * )

```

```

Begin

```

```

CluScr;

```

```

Write( File; ^);

```

```

ReadLn(Fname);

```

```

Write( Month; ^);

```

```

ReadLn(Month);

```

```

Write( Day; ^);

```

```

ReadLn(Day);

```

```

Write(Year: );
ReadLn(Hour)
Write(Hour);
Write(Minute: );
ReadLn(Second);

SetFileTimeAndDate(Fname,
                    Month, Day, Year,
                    Hour, Minute, Second,
                    Error);

GetFileTimeAndDate(Fname, Time—st, Day—st, Error);
If Error Then
    WriteLn(Error! )
Else
    WriteLn(Time—st, ', ', Day—st);
WriteLn;
Write(Press ENTER... );
ReadLn;
End.

```

当 GetFileTimeAndDate 调用 MsDos 时，它报告的时间放在寄存器 CX 中，日期放在 DX 中，然后被存入 Word 型的变量。随后这些变量自动地分解成各个成分：时，分，秒和年、月、日。这些成分组合成二个字符串通过参数 Time—St 和 Day—St 传递回来。

为了设置文件的时间和日期，我们必须首先计算表示时间和日期的数值，在 SetFileTimeAndDate 中，输入数值(时分秒，年月日)作为参数传递，过程中把它们转换成两个数，把时间装入寄存器 CX，把日期装入 DX，然后调用 MsDos 为文件设置时间和日期。

#### § 11.3.2.4 报告换档状态

Turbo pascal 不能直接读到某些 PC 上最高效的键：NUMLOCK，SCROLL LOCK，CTRL，ALT，两个换档键，CAPSLOCK 以及 INS。报告这些键状态的 BIOS 中断 16h 可以增加我们对键盘的控制。

为了检查这些特殊键的状态，使用中断 16h 并将 AH 寄存器置成 2。在中断 16h 完成后，它在 AL 寄存器中返回状态字节。该字节的每一位都指出八个特殊键之一的状态。

在下面的过程中，中断 16h 检查八个特殊键的状态：

```

Program Shift;
Uses CRT, DOS;
Var
    Ins,
    CapsLock, NumLock,

```

```

ScrollLock ;
Alt ,
Ctrl ,
LeftShift ,
RightShift : Boolean ;

```

```

( * * * * * )

```

```

Procedure ShiftStatus (Var Ins ,
                      CapsLock ,
                      NumLock ,
                      ScrollLock ,
                      Alt ,
                      Ctrl ,
                      LeftShift ,
                      RightShift : Boolean) ;

```

```

Var
  Regs : Registers ;

```

```

Begin

```

```

  Regs.AH := 2 ;

```

```

  Intr ($ 16 , Regs) ;

```

```

  RightShift := (Regs.AL And $ 01) > 0 ;

```

```

  LeftShift := (Regs.AL And $ 02) > 0 ;

```

```

  Ctrl := (Regs.AL And $ 04) > 0 ;

```

```

  Alt := (Regs.AL And $ 08) > 0 ;

```

```

  ScrollLock := (Regs.AL And $ 10) > 0 ;

```

```

  NumLock := (Regs.AL And $ 20) > 0 ;

```

```

  CapsLock := (Regs.AL And $ 40) > 0 ;

```

```

  Ins := (Regs.AL And $ 80) > 0 ;

```

```

End ;

```

```

( * * * * * )

```

```

Begin

```

```

  ClrScr ;

```

```

  Writeln ( 'Press Ins and then Ctrl to stop. . . ' ) ;

```

```

  Repeat

```



```

ShiftStatus(Ins,CapsLock,NumLock,ScrollLock,
            Alt,Ctrl,LeftShift,RightShift);

GotoXY(1,4);
WriteLn('Ins.....',Ins,' ');
WriteLn('CapsLock....',CapsLock,' ');
WriteLn('NumLock.....',NumLock,' ');
WriteLn('ScrollLock..',ScrollLock,' ');
WriteLn('Alt.....',Alt,' ');
WriteLn('Ctrl.....',Ctrl,' ');
WriteLn('LeftShift..',LeftShift,' ');
Until (Ins And Ctrl);

WriteLn;
Write(' Press ENTER... ');
ReadLn;
End.

```

ShiftStatus 过程为 8 个特殊键的每一个都设置了一个布尔参数。它检查返回在寄存器 AL 中的字节的每一位，使八个参数按照状态字节的各位分别被设置。

### § 11.3.3 使用 DOS 单元中的其它例程

虽然我们能用 Intr 和 MsDos 过程访问 BIOS 或 DOS 的各种功能，但 DOS 单元也包含许多使访问这些功能更容易的其它例程，这些例程在下文中将作一个简短的描述，它们的详细描述请参看附录。

#### 1. 中断支持例程

中断支持例程是我们访问任意 DOS 或 BIOS 服务或者按装我们自己的中断服务程序的工具，除了上一节介绍的 intr(执行任意中断服务)和 MsDos(仅执行 DOS 服务)两个过程之外，还有 GetIntVec，它返回特定中断的例程的当前地址；SetIntVec，它用我们自己的例程代替现有的中断例程。

#### 2. 日期和时间例程

系统时钟保持着当前日期和时间的轨迹。我们能 GetDate 和 GetTime 过程从系统时钟得到日期和时间，同样我们也能用 SetTime 和 SetDate 设置系统时钟的时间和日期。

文件有它们自己的时间戳记，这是一个含有文件建立或最近一次更新的时间和日期的长整数。我们能 GetFTime 得到任何文件的时间戳。在我们能解释文件的时间戳之前，我们必须让它通过 UnPacktime 以产生日期和时间。我们也能用 Packtime 完成相反的过程，它把时间和日期转化成一个我们能在 SetFTime 中用来设置一个文件的时间和日期的长整数。

#### 3. 磁盘和文件例程

DOS 单元包含两个磁盘状态例程：DiskFree 和 DiskSize。diskFree 返回磁盘的空闲字节量，DiskSize 返回磁盘上已用和未用字节的总量。

DOS 单元中最有用的例程是那些对磁盘文件进行操作的例程。Find First 和 Find next 让我们读到任意目录中的文件。Find First, 正如其名字所暗示的, 读目录中第一个文件的信息, 而 Find Next 得到后继文件的信息。在查找中我们可以使用 DOS 通配符(Wildcard Character), 甚至可以指定要查找的文件的类型(例如归档、系统、除藏等)。

如果我们想知道指定文件的属性, 我们可以用 Get Attr, 它返回给定名字的文件的文件属性字节。另一方面, Set Attr 允许我们设置文件的属性字节值。FExpand 函数有一个文件名作参数并返回完整的文件说明串, 包括磁盘驱动器、路径和文件名, FSplit 所做的正好相反, 它以完整的文件说明串为参数, 返回各个分量: 路径, 文件名和扩展名。FSearch 在一个目录表中寻找一个文件, 如果文件找到, FSearch 返回完整的文件说明串, 反之返回一个空串。

#### 4. 进程处理例程

Exec 和 Keep 是 DOS 单元中的两个最高级的例程。Exec 允许我们在一个程序里执行其它程序或 DOS 外壳(shell), 而 Keep 终止一个程序但保持它驻留内存。当使用 Exec 时, Swap Vectors 例程提供某种安全系数。当我们使用 Exec 在我们的 Turbo pascal 程序中运行一个子程序(进程)时, 我们就冒着子程序永久地更换了中断向量表的危险。在调用 Exec 之前和之后调用 Swap Vectors, 我们就可以确信中断向量会恢复到原先的状态。最后一个进程控制例程是 DosExitCode, 它提供程序结束时的 DOS 错误结果码, 这个代码可由其它程序或批文件使用。

DOS 维护着一个包含计算机的环境信息的内存单元。环境信息包括文件句柄的数目, COMMAND.COM 的位置等等, 这些信息包含在一组环境串(environment Stings)中。函数 EnvCount 返回内存中环境串的数目, 函数 EnvStr 返回指定的函数串, GetEnv 返回指定的环境要素(如 FILES, COMSPEC, PATH 等)的环境串。

最后, DOS 单元还有一组杂项例程用于提供信息和控制计算机。DosVersion 例程返回一个整数, 其高低字节包含所用 DOS 的版本号。个人计算机还有两个特性—Control—break 检查和 disk—Write 验证—可以设置成打开或关闭。CTRL—BRAEK 组合键用于终止程序。当 Control—break 检查关闭时, DOS 仅在控制台、打印机或 I

O 通讯期间检查 CTRL—BREAK; 反之, DOS 在每个系统用中检查 CTRL—BREAK。GetCBreak 例程返回指示 Control—break 检查打开还是关闭的一个布尔值, SetCBreak 接受一个设置 Control—break 打开或关闭的布尔值。类似地, GetVerify 例程返回指示 disk—write 验证打开还是关闭的布尔值, 而 SetVerify 接受设置 disk—write 验证打开或关闭的布尔值。当 disk—write 验证启动, DOS 验证每一次磁盘写, 当它关闭, DOS 不执行验证。

在下面程序中说明了以上每个例程(除了 Keep)的用法, 这将给我们提供一个如何在我们的程序中使用它们的良好思路。

```
{ $F+ }
{ $M 10000, 0, 0 }
Program Test;
Uses DOS, CRT;
Var
  OldTimerVec : Pointer;
  Regs : Registers;
  ClockFag : Word;
```

```

x,y : Byte;
i : Integer;
S : String;

```

```

( * * * * * )

```

```

Procedure XYChar(x,y : Byte;
                 c : Char;
                 fg,bg : Byte);

```

```

{
This routine writes a character to the screen using
a BIOS routine.
}

```

```

Begin
GotoXY(x,y);
FillChar(Regs,SizeOf(Regs),0);
With Regs Do
  Begin
    AH := $09;          (* Call BIOS Service 9      *)
    AL := Ord(C);        (* Character in AL      *)
    BH := 0;             (* Video page in BH     *)
    BL := (bg shl 4) + fg; (* Attribute in BL      *)
    CX := 1;             (* Number of repetitions in CX *)
  End;
Intr($10,Regs);
End;

```

```

( * * * * * )

```

```

Procedure Clock; Interrupt;
{
This procedure replaces the original clock interrupt.
}
Begin
CallOldInt(OldTimerVce); (* Call the original routine *)
Inc(ClockFlag,1); (* Increment the counter *)
Str(ClockFlag,S); (* Convert to string *)
For i := 1 To Length(S) Do (* Write string with XY/char *)
  XYChar(i,10,s[i],Yellow,Black);

```

End;

( \* \* \* \* \* )

Function PadRight(S : String; L : Word) : String;

{

This function adds blank characters to  
a String until it reaches length L. If  
is longer than L to begin with,  
this function truncates the String.

}

Begin

If Length(S) > L Then

  S[0] := Chr(L);

While Length(S) < L Do

  S := S + ' ';

PadRight := S;

End;

( \* \* \* \* \* )

Procedure InerruptSupportDemo;

Var

  I : Word;

  FileName : String;

  Regs : Registers;

Begin

ClrScr;

( \* Use BIOS interrupt to determine video adapter \* )

FillChar(Regs,SizeOf(Regs),0);

Regs.AH := \$0F;

Intr(\$10,Regs);

  Case Regs.AL of

    1..6 : WriteLn('CGA');

    7 : WriteLn('Monochrome');

    8..10 : WriteLn('EGA');

  End;

( \* Use DOS service to determine if file is read-only \* )

WriteLn;

Write( 'Enter file name: ');

ReadLn(FileName);

FillChar(Regs,SizeOf(Regs),0);

( \* Add null character to String \* )

( \* so String can be used as \* )

( \* ASCIIZ String. \* )

FileName := FileName + #0;

With Regs Do

Begin

AH := \$43;

DS := Seg(FileName);

DX := OfS(FileName) + 1; ( \* Skip length byte \* )

End;

MsDos(Regs);

If (Regs.CL And \$01) > 0 Then

WriteLn( 'File is Read-Only' )

else

WriteLn( 'File is not Read-Only' );

WriteLn;

Write( 'Press ENTER... ');

ReadLn;

ClockFlag := 0;

GetInVce(8,OldTimerVce); ( \* Save interrupt address \* )

SetIntVce(8,&Clock); ( \* Point time interrupt to Clock \* )

WriteLn;

Write( 'Press ENTER... ');

ReadLn;

SetIntVce(8,OldTimerVce); ( \* Restore interrupt address \* )

End;

( \* \* \* \* \* )

Procedure DateAndTimeProcedures;

Const

DayName: Array [0..6] of String[10] = ( 'Sunday',  
Monday',  
Tuesday',  
Wednesday',  
Thursday',  
Friday',  
Saturday');

Var

Year, Month, Day, DayOfWeek,  
Hour, Minute, Second, Sec100 : Word;  
fname : String;  
f : File;  
T : LongInt;  
DT : DateTime;

Begin

ClrScr;

(\* Display current date and time \*)

WriteLn('Current date and time.');

GetDate(Year, Month, Day, DayOfWeek);

WriteLn('System time = ', Hour, ':', Minute, ':', Second, ':', Sec100);

WriteLn;

WriteLn('Set current date and time.');

(\* Set new date and time \*)

Write('Enter year (1980 Or later):');

ReadLn(Year);

Write('Enter month:');

ReadLn(Month);

Write('Enter day:');

ReadLn(Day);

Write('Enter hour:');

ReadLn(Minute);

Second := 0;

Sec100 := 0;

SetDate(Year, Month, Day);

SetTime(Hour, Minute, Second, Sec100);

( \* Display new date and time \* )

WriteLn( 'New date and time. ' );

GetDate(Year, Month, Day, DayOfWeek);

WriteLn( 'System Date = ', DayName[DayOfWeek], ', ', Month, '/', Day, '/', Year );

GetTime(Hour, Minute, SEcond, Sec100);

WriteLn( 'System time = ', Hour, ': ', Minute, ': ', Second, ': ', Sec100 );

writeLn;

Write( 'Press ENTER... ' );

ReadLn;

( \* Get time and date for a file \* )

ClrScr;

WriteLn( 'Get date and time for a file. ' );

Write( 'Enter file name: ' );

ReadLn(fname);

Assign(f, fname);

Reset(f);

GetFTime(f, T);

UnPackTime(T, DT);

With DT Do

Begin

WriteLn( 'File name: ', fname );

WriteLn( 'Date: ', Month, '/', Day, '/', Year );

WriteLn( 'Date: ', Hour, ': ', Min, ': ', Sec );

End;

( \* Set new time and date for file \* )

WriteLn( 'Set file's date and time. ' );

With DT Do

Begin

Write( 'Year: ' );

ReadLn(Year);

Write( 'Month: ' );

ReadLn( Month );

Write( 'Day: ' );

ReadLn( Day );

Write( 'Hour: ' );

```

    ReadLn(Jpir);
    Write( Minute: ^);
    ReadLn(Minute);

    Second := 0;
    End;
PackTime(DT,t);
SetFTime(f,T);
UnPackTime(T,DT);
With DT DO
    Begin
        Writeln( File name: ^,fname);
        Writeln( Date:      ^,Month,/^,Day,/^,Year);
        Writeln( Date:      ^,Hour,^;^,Min,^;^,Sec);
    End;

Close(f);
Writeln;
Write( Press ENTER... ^);
ReadLn;
End;

( * * * * * )

Procedure DiskStatusFunctions;
Var
    S : LongInt;
Begin
    ClrScr;
    Writeln( Disk status. ^);
    S := DiskFree(0);
    Writeln( Free space on disk = ^,s,^ bytes/^, Div 1024, ^ Kbytes ^);
    S := DiskSize(0);
    Writeln( Total space on disk = ^,s,^ byte/^,s Div 1024, ^ Kbytes ^);

    Writeln;
    Write( Press ENTER... ^);
    ReadLn;
End;

```



( \* \* \* \* \* )

Procedure FileHandling;

Var

Atr : Word;

f : File;

S : String;

DT : DateTime;

Srec : SearchRec;

PS : PathStr;

DS : DirStr;

FN : NameStr;

End : ExtStr;

Begin

ClrScr;

WriteLn( Press ENTER for a directory listing. );

ReadLn;

FindFirst( '\*.\*', AnyFile, Srec );

While DosError = 0 Do

Begin

With Srec Do

Begin

UnPackTime( /time, DT );

With DT Do

Begin

S := FExpand( Name );

Fsplit( S, Ds, FN, EN );

WriteLn( PadRight( DS, 10 ), ' ',

PadRight( FN, 9 ), ' ',

PadRight( EN, 5 ), ' ',

Size: 7, ' ', Year );

End;

End;

FindNext( Srec );

End;

```

WriteLn;
Write( 'Press ENTER... ');
ReadLn;

WriteLn( 'Search for a file by name. ');
ClrScr;
Repeat
  Write( 'Enter file name: ');
  ReadLn(DS);
  S := Fsearch(S,DS);
  If S = '' Then WriteLn( '/file not found... ');
Until S > '';

Assign(f,S);
GdtFAttr(f,Attr);

If ((Attr And ReadOnly) > 0) Then
  WriteLn( 'File is read only')
Else
  WriteLn( 'File is not read only');

If ((Attr And Hidden) > 0) Then
  writeLn( 'File is hidden')
Else
  WriteLn( 'File is not hiddn');

If ((Attr And SysFile) > 0) Then
  WirteLn( 'File is system file')
Else
  WriteLn( 'File is not system file');

If ((Attr And VolumeID) > 0) Then
  WriteLn( 'File is Directory')
Else
  WriteLn( 'File is not Directory');

If ((Attr And Archive) > 0) Then
  WriteLn( 'File is Directory')

```

```

Else
    WriteLn( File is not Directory );

If ((Attr And Archive) > 0) Then
    WriteLn( File is Archive )
Else
    WriteLn( File is not Archive );

WriteLn;
Write( Press ENTER... );
ReadLn;
End;

( * * * * * )

Procedure ProcessHandling;
Begin
    ClrScr;
    WriteLn( ' DOS SHELL: Type EXIT to return to program... ');

    SwapVectors;
    Exec(GetEnv( 'COMSPEC' ), ' ');
    WriteLn( DosExitCode = ', DosExitCode);
    SwapVectors;
    WriteLn;
    Write( Press ENTER... );
    ReadLn;
End;

( * * * * * )

Procedure EnvironmentHandling;
Var
    i : Integer;
Begin
    ClrScr;
    WriteLn( Environment info: );
    WriteLn( Number of Environment Strings = ', EnvCount);
    For i := 1 To EnvCount Do
        WriteLn(i, 2, ', ', EnvStr(i));

```

```

WriteLn;
WriteLn('COMSPEC = ' + GetEnv('COMSPEC'));
WriteLn;
Write(' Press ENTER... ');
End;

```

```

( * * * * * )

```

```

Procedure MiscProcs;

```

```

Var

```

```

    YN : Char;

```

```

    DosVer : Word;

```

```

    Verify,

```

```

    CBreak : Boolean;

```

```

Begin

```

```

    ClrScr;

```

```

    WriteLn('Other information. ');

```

```

    DosVer := DosVersion;

```

```

    WriteLn('DOS Version: ', Lo(DosVer), ', ', Hi(DosVer));

```

```

    WriteLn

```

```

    GetCBreak(CBreak);

```

```

    If CBreak Then

```

```

        Begin

```

```

            WriteLn('Ctrl-Break checking is On. ');

```

```

            Write(' Turn Ctrl-Break Checking OFF? Y/N: ');

```

```

            ReadLn(YN);

```

```

            If UpCase(YN) = 'Y' Then

```

```

                SetCBreak(FALSE);

```

```

            End

```

```

    Else

```

```

        Begin

```

```

            WriteLn('Ctrl-Break checking is OFF. ');

```

```

            Write(' Turn Ctrl-Break checking ON? Y/N: ');

```

```

            ReadLn(YN);

```

```

            If UpCase(YN) = 'Y' Then

```

```

                SetCBreak(TRUE);

```

```

            End;

```

```

    GetVerify(Verify);

```

```

If Verify Then
    Begin
        WriteLn( 'Disk write verification is On. ');
        Write( 'Turn disk write verification OFF? Y/N: ');
        ReadLn(YN);
        If UpCase(YN) = 'Y' Then
            SetVerify(FALSE);
        End
    Else
        Begin
            WriteLn( 'Disk write verify is OFF. ');
            Write( 'Turn disk write verification ON? Y/N: ');
            ReadLn(YN);
            If UpCase(YN) = 'Y' Then
                SetVerify(TRUE);
            End;
        End;

    WriteLn;
    GetCBreak(CBreak);
    If CBreak Then
        WriteLn( 'Ctrl-Break checking is ON. ');
    Else
        WriteLn( 'Ctrl-Break checking is OFF. ');

    SetVerify(Verify);
    If Verify Then
        WriteLn(Disk write verification is ON. ');
    Else
        WriteLn( 'Disk write verify is OFF. ');

    WriteLn;
    Write( 'Press ENTER... ');
    ReadLn;
    End;

( * * * * * )

Begin
DateAndTimeProcedures;
DiskStatusFunctions;

```

```

FileHandling;
EnvironmentHandling; MiscProces;
InterruptSupportDemo;
ProcessHandling;
End.

```

## § 11.4 硬中断：远程通信与 TSR 实现

前面我们介绍了 DOS 的 BIOS 中断和 DOS 调用，它们属于软件中断，从本质上说，它们只是一组可以被有意识地调用的服务例程，只是利用了中断机制来处理调用控制的转移，并不是真正意义上的中断。从 § 11.2 节我们知道，中断是为了不漏掉任何要处理的外部事件，又不必经常不断地去查询是否发生了外部事件而提出的一种提高计算机效率的机制，就象电话的振铃系统，有呼叫时会响，平常就很安静，有时一整天都没有电话，有时却接二连三响个不停。换句话说，中断是为了对外部事件作出及时响应，而外部事件是不可预期的，但是软中断是一种系统服务，程序何时需要系统服务是事先知道的。因此本节着重介绍响应外部不可预期事件的硬中断处理。

典型的硬中断有不可屏蔽的电源故障中断，以及内部时钟中断、键盘中断、串行口中断、软盘驱动器中断等等，参见 § 11.2 中给出表 11-1。

硬中断的处理机制与软中断是一致的，无论是键盘中断还是内部时钟中断，一旦发生，计算机首先要保存当前程序运行的现场，然后从中断向量表中取出中断处理例程的入口地址，转到中断例程处理后，再恢复现场继续当前程序的运行。

与上节的讨论方式不同，上节讨论软中断主要着重于中断处理的功能，即提供的服务，本节则主要侧重于如何编写中断处理例程来完成所需要的功能，即替代系统提供的硬中断服务。编写中断例程比利用中断服务要复杂，尤其是硬件中断的不可预期性，使中断例程的执行与当前程序完全是异步并发的，因此编写中断例程就要十分小心不能对当前执行的任何程序产生副作用。本节中讨论远程通信和 TSR 实现时都会反复说明这一点。

### § 11.4.1 编写中断处理程序

我们知道，每次启动计算机，DOS 都会自动用标准中断例程的地址填写中断向量表，但是只要计算机开始运行，程序员就可以修改中断向量表的内容，使它们背面程序员自己写的中断处理过程。通常程序员很少修改软中断的地址，但对修改硬中断的处理程序，进而修改硬中断的向量地址表却是极有兴趣的，通过这种修改，往往可以完成许多奇妙的功能，例如 SuperKey 和 SideKick 一类内存驻留程序就是借助于这种修改才可能工作的。一个内存驻留程序可以通过改变键盘中断的地址而使击键到达主程序之前被截获并赋予新的解释。

中断地址的修改有两种方法：直接修改中断向量表或用 DOS 服务来修改。要直接修改内存，必须覆盖中断向量表的地址数组，象下面这样：

```

Var
  AsyncVector: Pointer;
  InterruptVector: Array[0..$FF] of Pointer
  Absolute $0000: $0000;

```

在这个例子代码中使用了 `Pointer` 数据类型, `Pointer` 是一个4字节的数据类型, 可以包含构成内存地址的段地址和偏移量。变量 `InterruptVector` 是一个256(0h 例 \$FFh)个地址的数组, 它被说明成绝对地处于 RAM 的最始端, 即中断向量表驻留的位置。既然 `InterruptVector` 与表共享相同的空间, 那么当我们修改其中一个的值时, 也就修改了另一个的值。为了修改中断向量表中的地址, 使用下面的语句:

```
AsyncVector := InterruptVector [$0C];
```

```
InterruptVector[$0C] := @AsyncInt;
```

第一个语句将中断12(十六进制的 0C)的向量存放在变量 `AsyncVector` 中, 保存原始地址是重要的, 因为最终需要恢复它。第二个语句用 Turbo Pascal 的 `AsyncInt` 过程的地址来替代原向量。

Turbo Pascal 操作符 `@` 返回 `AsyncInt` 过程的 4 字节地址(段地址和偏移量)。然后地址装入中断向量表。

在使用该中断向量之后, 我们必须恢复保存在指针型变量 `asyncVector` 中的原地址, 如果不恢复这个地址, 也许会使计算机崩溃, 而不得不重新引导(reboot)。下面这一行代码就可以使我们恢复原先的中断向量地址

直接修改中断地址是简单的, 但使用 DOS 服务则更安全, 也适用于更广泛的计算机。DOS 服务 35h 报告一个向量的内容, 服务 25h 则更改中断向量表中的一个地址。很幸运, Turbo Pascal 的 DOS 单元提供了两个例程—`GetIntVec` 和 `SetIntVec`—可以准确地完成以上服务。这两个例程都需两个参数: 中断号(例如: 0Ch)和一个指针变量。保存一个中断向量以及用新向量替代它的处理如下所示: `GetIntVec($0C, AsyncVector);`

```
SetIntVec($0C, @AsyncInt);
```

在我们用毕该中断时, 用以下语句恢复原中断向量:

```
SetIntVec($0C, AsyncVector);
```

讨论完中断向量的更改替代之后, 就可以进一步讨论中断处理程序的编写。

中断处理程序是指作为中断的结果要执行的代码, 在前面的例子中, 处理程序是 Turbo Pascal 的叫做 `AsyncInt` 的过程, 它依附于中断 0Ch。中断处理程序不同于其它类型的过程, 因为我们并不总是知道它们会在什么时候执行, 例如中断 0Ch 在 COM1 上准备好一个字节时触发, 我们无法知道这会在什么时候发生, 因此我们的中断处理程序就必须格外小心以避免对计算机正常处理的干扰。换句话说, 一个设计良好的中断处理程序会在我们很少注意它的时候进入, 偷走一些计算时间, 然后不留痕迹地离开。

写中断处理程序的技巧是不留痕迹地离开。为了做到这一点, 处理程序在执行前必须保存所有的 CPU 寄存器并在处理完后恢复它们(处理程序不必保 CS, IP 或标志寄存器, 因为 DOS 在调用中断之前已经为我们保存好了)。但是应单单保存 CPU 寄存器是不够的, 我们还必须克服数据段带来的问题。

在调用中断处理程序的时候, 只有代码段(CS)和指令指针(IP)寄存器能设置成我们的处理程序需要的值, 换句话说, 包含当前被中断过程所用数据的数据段(DS)可能不包含中断处理程序要用的正确数据。如果不更正数据段, 我们的处理程序无法引用它自己的全局变量, 因此我们的中断处理程序必须在代码段中存放着自己的 DS 值, 以便能设置正确的 DS。

虽然这些看上去有点吓人, 但 Turbo Pascal 会来救我们。Turbo Pascal 使我们写中断处理程序时可以自动地保存所有寄存器, 建立 DS 寄存器并在中断处理完成时恢复所有寄存器。为

了说明一个过程是中断处理程序，我们只需在过程说明的尾部附加 Interrupt 指示符即可，如下所示：

```
Procedure IntHandler; Interrupt;
```

Interrupt 伪指令指示 Turbo Pascal 插入合适的代码以保存寄存器，建立 DS 寄存器，恢复寄存器以及在过程结束时发生 IRET 指令。我们可以在处理程序中有选择地说明的一个表示 CPU 寄存器的伪变量表：

```
Procedure IntHandler(Flags, CS, IP, AX, BX, CX, DX, SI, DI, DS, ES, BP);
```

```
Interrupt;
```

```
Procedure IntHandler(SI, DI, DS, ES, BP); Interrupt;
```

正如我们在上述列表中看到的，参数表可以包括所有 CPU 寄存器(Flags 到 BP)或寄存器的一个子集(例如，SI 到 BP)。可是有两点限制，第一，不能改变参数的顺序，第二，如果删除参数，只能从左边删除，例如，如果我们只想要 SI 和 DI 寄存器，那么我们也必须在参数表中包括 DI, DI, DS, ES 和 BP，次序也不能变。

在我们的中断程序执行时，我们可以访问所有我们列于过程头上的伪参数，这些参数的值是中断前 CPU 寄存器的内容。

注意 SP 和 SS 寄存器不包括在伪参数表中，这意味着我们的中断处理程序正用着被中断程序的栈。这有点冒险，因为我们完全不知道被中断程序的栈有多大。最安全的办法是在我们的中断处理程序中尽可能避免使用堆栈。下一节将说明中断处理程序如何用于建立远程通讯程序。

#### § 11.4.2 PC 远程通信及程序

远程通信是指数据从一台计算机通过电话线传输到另一台计算机。这从概念上来说简单的，但在实现上却很复杂。在概念上，计算机发送一个字节到串行口，串行口再将它发送到调制解调器，调制解调器将每个比特位译成音频信号并发送到另一台调制解调器，这台调制解调器再将音频信号译回比特位。译好的位送到接收的串行口，在那里这些比特位等着被软件收集组成字节数据。

在实现上，写一个程序来做所有这些事并不那么简单。复杂性部分来自我们必须控制的硬件元素的数目：RS-232 串行口，INS8250 通用异步接收传输器(UART)，8259 中断控制器以及调制解调器本身。另外，还有串行口所用的许多种不同的调制解调器速度、奇偶性、停止位，数据位等等。

下面的程序以做 Async Communications。它用中断 12 提供简单的远程通信能力。程序假定我们的第一串行口连接着一个 Hayes 兼容的调制解调器。在我们运行程序之前，Select-ModemSet 过程设置我们想要调用的远程计算机的一些数值(波特率、数据位、停止位、奇偶性等)。

```
Program CommX;
```

```
{ $s-, R-, I-, V-, F+ }
```

```
Uses CRT, DOS;
```

```
Type
```

```
ModemSetType = Record
```



```

    ComPort,
    BPS,
    DataBits,
    StopBits      : Integer;
    Parity        : Char;
    PhoneNumber    : String;
End;

Const
    MainDseg : Integer = 0;
    MaxBufLen  = 1024;
    CR         = #13;

    ( * * * * * )
    ( * ISN8250 UART Registers. * )
    ( * * * * * )
    DataPort = $03F8;
    ( * Contains 8 bits to transmit or receive. * )
    IER       = $03F9;
    ( * Enables the serial port when set equal to 1. * )
    LCr       = $03FB;
    ( * Sets communications parameters. * )
    MCR       = $03FC;
    ( * Bits 1, 2 and 4 are turned on to ready modem. * )
    LSR       = $03FD;
    ( * When bit 6 is on, it is safe to send a byte. * )
    MDMMSR    = $03FE;
    ( * Initialized to 80h when starting.

ENBLRDY     = $01; ( * Initial value for Port[IER]. * )
MDMMOD      = $0B; ( * Initial value for Port[MCR]. * )
MDMCD       = $80; ( * Initial value for Port[MDMMSR]. * )

INTECTLR = $21; ( * Port for 8259 Interrupt Controller. * )

Var
    ModemSet      : ModemSetType;
    AsyncVectior  : Pointer;
    Regs          : Registers;
    Buffer         : Array [1..MaxBufLen] Of Char;

```

```

1.
CharsInBuf,
CircOut,      : Word;
Orig          : Char;

```

```

( * * * * * )

```

```

Procedure ClearBuffer;
Begin
  CircIn := 1;
  CircOut := 1;
  CharsInBuf := 0;
  FillChar(Buffer, SizeOf(Buffer), 0);
End;

```

```

( * * * * * )

```

```

Procedure SelectModemSet;
Begin
  With ModemSet Do
    Begin
      ComPort := 1;      (* Must be 1 in this program. *)

      Repeat
        Write( BPS (300,1200); ^);
        ReadLn(BPS);
      Until (BPS = 300) Or (BPS = 1200);

      Repeat
        Write( Data bits (7,8); ^);
        ReadLn(DataBits);
      Until DataBits in [7,8];

      Repeat
        Write( Stop bits (0,1); ^);
        ReadLn(StoBits);
      Until StipBits in [0,1];

      Repeat
        Write( Parity (N,E,O); ^);

```

```

    ReadLn(Parity);
    Parity := UpCase(Parity);
    Until Parity in ['N', 'E', 'O'];

    Write('Phone number: ');
    ReadLn(PhoneNumber);
    End;
End;

( * * * * * )

```

```

Procedure AsyncInt; Interrupt;
Begin
    Inline($FB);    (* STI *)

    If (CharsInBuf < MaxBufLen) Then
        Begin
            Buffer[CircIn] := Char(Port[DataPort]);
            If (CircIn < MaxBufLen) Then
                Inc(CircIn, 1)
            Else
                CircIn := 1;
            Inc(CharsInBuf, 1);
        End;
    End;

```

```

    Inline($FA);    (* CLI *)

```

```

    Port[$20] := $20;
End;

```

```

( * * * * * )

```

```

procedure SetSerialPort(ComPort,
                        BPS,
                        SopBits,
                        DataBits: Integer;
                        Parity : Char);

Var
    Regs : Registers;
    parameter : Byte;

```

```

Begin
  Case BPS Of
    300 : BPS := 2;
    1200 : BPS := 4;
  End;

  If StopBits = 2 Then
    StopBits := 1
  Else
    StopBits := 0;

  If DataBits = 7 Then
    DataBits := 2;
  Else
    DataBits := 3;

  Parameter := (BPS Shl 5) + (StopBits Shl 2) + DataBits;

```

```

  Case Parity Of
    'E' : Parameter := Parameter + 8;
    'O' : Parameter := Parameter + 24;
  End;

```

```

With Regs Do
  Begin
    DX := ComPort - 1;
    AH := 0;
    AL := Parameter;
    Flags := 0;
    Intr($14, Regs);
  End;

```

```
End;
```

```
( * * * * * )
```

```
Procedure EnablePorts;
```

```
Var
```

```
  B : Byte;
```

```
Begin
```

```

MainDseg := Dseg;
ClearBuffer;
GetIntVec($OC,@AsyncInt);

```

```

B := Port[INTCTLR];
B := B And $OEF;
Port[INTCTLR] := B;
B := Port[OCr];
B := B And $7F;

```

```

Port[LCR] := B;
Port[IER] := ENBLRDY;
Port[MCR] := $08 Or MODMMOD;
port[MDMMSR] := MDMCD;
Port[$20] := $20;
End;

```

( \* \* \* \* \* )

```

Function GetCharInBuf : Char;
Begin
If CharsInBuf > 0 Then
    Being
    GetCharInBuf := Buffer[(CircOut)];
    Inc(CircOut,1)
Else
    CircOut := 1;
    Dec(CharsInBuf,1);
End;
End;

```

( \* \* \* \* \* )

```

Function CarrierDetected : Boolean;
Var
    Ch : Char;
    Timer : Integer;
    CarrierDetected := False;
    Timer : Integer;
Begin

```

```

CarrierDetected := False;
Timer := 40;
While (port[MDMSR] And $80) <> $80 Do
  Begin
    If KeyPressed Then
      Begin
        Ch := ReadKey;
        If Ch = #27 Then
          Exit;
        End;
      End;
    If (CharsInBuf > 0) Then
      Begin
        Ch := GetCharInBuf;
        Write(Ch);
        If Ch = CR Then
          WriteLn;
        End;
      End;
    If Timer = 0 Then
      Exit
    Else
      Begin
        Dec(Timer, 1);
        Delay(1000);
      End;
  End;
CarrierDetected := True;
End;

```

( \* \* \* \* \* )

```

Procedure SendChar(B: Byte);
Begin
  While ((Port[LSR] And $20) <> $20) Do
    Begin
      End;

```

```

Port[Dataport] := B;
End; ( * * * * * )

```

```

Procedure StringtoPort(S : String);

```

```

Var

```

```

    I : Integer;
Begin
For I := 1 to Length(S) Do
    SendChar(Ord(S[I]));
SendChar(Ord(CR));
End;

( * * * * * )

```

```

Procedure DisablePorts;

```

```

Var

```

```

    B : Byte;

```

```

Begin

```

```

    ( * Turn off carrier signal. * )

```

```

StringToPort('ATCO');

```

```

    ( * Turn off the communication interrupt for COM 1. * )

```

```

B := Port[LCR];

```

```

B := B And $7F;

```

```

Port[LCR] := B;

```

```

Port[IER] := $0;

```

```

    ( * Disable OUT 2 on 8250. * )

```

```

Port[MCR] := $0;

```

```

Port[$20] := $20;

```

```

SetIntVec($0C, AsyncVector);

```

```

MsDos(Regs);

```

```

End;

( * * * * * )

```

```

Function SuccessfulConnect(PhoneNumber : String) : Boolean;

```

```

Var

```

```

    S : String;

```

```

Begin

```

```

    ( * ATDT assumes touch-tone dial. * )

```

```

S := 'ATDT' + PhoneNumber;

```

```

StringToPort(S);

```

```

Delay(300);
If CarrierDetected Then
SuccessfulConnect := True
Else
  Begin
    Write( Error; Unable To Connect. );
    StringToPort( 'ATCO' ); ( * Turn off carrier signal. * )
    SuccessfulConnect := False;
  End;
End;

```

( \* \* \* \* \* )

Procedure SetHayesModem;

Begin

```

StringToPort( 'ATCO' );      ( * Turn off carrier signal.
Delay(1000);                  ( * Wait a second.          * )
StringToPort( 'ATZ' );      ( * Reset modem to cold--start.  * )
Delay(1000);                  ( * Wait a second.          * )
StringToPort( 'ATF1' );     ( * Full Duplex.              * )
Delay(1000);                  ( * Wait a second.          * )
StringToPort( 'ATEO' );     ( * Do not echo in command    * )
                             state.
Delay(1000);                  ( * Wait a second.          * )
StringToPort( 'ATV1' );     ( * Verbal result codes.    * )
Delay(1000);                  ( * Wait a second.          * )
StringToPort( 'ATQO' );     ( * Send result codes.      * )
Delay(1000);                  ( * Wait a second.          * )
End;

```

( \* \* \* \* \* )

Procedure StartCommunicating;

Var

OutChar,

InChar : Char;



```

Begin
ClearBuffer;

Repeat
If (CharsInBuf > 0) Then
Begin
InChar := GegCharInBuf;
If InChar in [# 32..# 126] Then
Write(InChar)
Else
WriteLn;
End;
If KeyPressed Then
Begin
OutChar := ReadKey;
If (OutChar <> # 27) Then
Begin
SendChar = CR Then
WriteLn
Else
WriteLn
Else
WriteLn
Else
Write(OutCahr);
End;
End;
Until OutChar = # 27;
End;

( * * * * * )

Begin
ChuSer;

With ModemSet;

With ModemSet Do
SetSerialPort(ComPort,BPS,StopBits,DataBits,Parity);

EnablePorts;

```

```
SetHayesModem;
```

```
If SuccessfulConnct (ModemSet, PhoneNumber) Then
```

```
    Start/communicating;
```

```
WriteLn;
```

```
Write( 'Logging off. . . ');
```

```
StringToPort('ATZ');          (* Reset mode to cold -- start.      *)
```

```
Delay(1000);                   (* Wait a second.              *)
```

```
DisablePrts;
```

```
End.
```

这个程序的主体调用几个过程和函数，并给出了建立步通信的步骤。第一个被调过程是 SelectModemSet，它允许用户指定要用的通讯口，每秒位数，停止和数据位数，奇偶性和电话号码。（注意该程序设计成只能用通讯口 1 (COM1)。这些数据项存放在类型为 ModemSetType 的记录中，记录定义如下：

```
Type
```

```
    ModemSetType = Record
```

```
        Comport,
```

```
        Bps,
```

```
        DataBits,
```

```
        StOPBits: Integer;
```

```
        Partiy : Char;
```

```
        PhoneNumber: String;
```

```
    End;
```

这个记录初始化之后，内容传到 SetSerialPort 过程，它使用 BIOS 中断 14h 将串行口设置成给定的参数。为了使用这个中断，我们必须把 AH 置成 0，以告诉 BIOS 是初始化串行口；把 AL 置为一个参数字节，其各个位包含通讯设置；并且对于 COM1 把 DX 置为 0，对于 COM2 把 DX 置为 1。

使用中断 14h 最困难的部分是设定参数字节的位。表 11-18 中列出了该字节中每一位的定义。

		Bit	
Parameter	Bits Used	Pattern	Meaning
Data Bits	Bits 0-1	00	5 data bits
		01	6 data bits
		10	7 data bits
		11	8 data bits

Stop /bits	Bit 2	0	1 stop bit
		1	2 stop bits
Parity	Bits 3—4	00	No parity
		01	Odd parity
		10	No parity
		11	Even parity
Speed (BPS)	Bits 5—7	000	100 BPS
		001	150 BPS
		010	300 BPS
		011	600 BPS
		100	1200 BPS
		101	2400 BPS
		110	4800 BPS
		111	9600 BPS

表11-8 调用 BIOS 中断 14h 时 AL 寄存器的内容

一旦设置完调制解调器，就由过程 Enable Ports 来安装中断，它先保存旧中断向量地址，然后安装 AsynchrInt 过程的地址，再使 INS 8250 UART 芯片准备通讯。

准备工作的最后一步是将调制解调器用 SetHayesModem 过程初始化成合适的设置。虽然存在许多可用的调制解调器，但 Hayes Smartmodem 是个人计算机通常接受的标准设备。这里所用的调制解调器命令应该可以在任何与 Hayes Smartmodem 兼容的调制解调器中使用。表 11-19 列出了 Hayes Smartmodem 上有效的命令。Hayes Smartmodem 的内部操作的完整解释，请参见 Hayes Microcomputer Products, Inc 的“Smartmodem 1200 硬件参考手册”。如果我们使用一个不兼容的调制解调器，那就修改这个过程以适应我们的调制解调器。

命令	参数	描述
A	无	调制解调器不等待振铃，回答电话呼叫。 这用于从声音方式 到数据方式的改变。
A	无	重复上一条命令
Cn	0,1	传输器关闭。当 n=1，调制解调器呼叫、 回答或连接到另一调制解调器。指有其它时刻，n=0

, (逗号)	无	当拨电话号码时, 产生两秒钟的延时。
Dn	数码	调制解调器进入发报状态, 并拨由 S 表示的电话号码
En	0, 1	当 n=0, 处于命令状态的调制解调中不回送字符。 当 n=1 时回送字符。
Fn	0, 1	当 n=0, 调制解调器按半双工工作; 当 n=1, 调制解调器按全双工工作
Hz	0, 1, 2	此命令控制我们的电话的拨号音调, 当 n=0, 调制解调器“挂机”, 没有拨号音出现。 当 n=1, 调制解调器“摘机”, 拨号音出现。 参数 2 用于使用业余无线电设备的特殊应用。 该命令请求 Smartmodem 的三位数字产品码。 前两位数字指出产品 (例如, 12 指示 Smartmodem 1200), 第三位数字表示修改号。
In	0, 1	M 命令控制扬声器, 当 n=0, 扬声器关闭, 当 n=1, 扬声器 打开直到检测到载波。 当 n=2, 扬声器始终打开。
O	无	当调制解调器联机 (on-line) 时, O 命令使之回到命令状态 在命令状态, 任何调制解调器命令均能发出。
P	无	告诉调制解调器用脉冲而不是用音调拨号。 调制解调器能送出报各其状态的结果码。
Qn	0, 1	命令 q0 告诉调制解调器发出状态码, Q1 关闭此特性。
R	无	当呼叫一个只发报调制解调器时, 将 R 置于电话号码尾部
Sr?	1..16	读由 r 指定的调制解调器的 16 个寄存器之一的内容
Sr=n	r=1..6	将调制解调器寄存器 r 的值置为 n
;	无	将分号 (;) 置于拨号命令之后强制调制解调器 在与远程调制解调器连接之后回到命令状态
T	无	告诉调制解调器用音调而不是用脉冲拨号。
Vn	0, 1	调制解调器能用数和词返回码字。 Vo 选择数字码, V1 选择 词。
Xn	0, 1	调制解调器能返回码字的基本集或扩展集。 Xo 选择基本集, X1 选择扩展集。

表11-19 Hayes Smartmodem 命令

现在串行口设置好了, 中断安装了, 调制解调器也初始化了, 可以开始通信了。布尔函数 Successful Connect (Modem Set, Phone Number) 将电话号码拨出, 并等待来自调制解调器的载波检测信号, 这个信号指示连接已建立, 函数返回 TRUE 时, 通信开始。如果没有得到载波检测信号, 函数返回 FALSE 并且程序结束。

载波检测信号收到以后, 控制传送给过程 Smart Communicating, 它传输我们打入的字符, 并显示从远程计算机收到的字符。这个过程持续到我们按下 ESC 键, 该键产生一个 ASCII 码 27。

最后一步, DisablePorts 过程在中断向量表中安装原始中断地址并复位 UART 芯片。

AsyncCommunications 程序还用到一个称为循环输入缓冲器的数据结构及其相关函数。循环输入缓冲器是中断驱动通信程序的核心要素之一。因为数据可能在任何时候到达串行口, 中断处理程序必须在计算机做其它事时也能捕捉并处理到达的数据。如果中断不将字符存在一个缓冲器中, 那么在程序有时间来捕捉它之前就可能丢失。循环缓冲器是一个字符数组, 它通过暂时存贮字符直到计算机从输入字符流中取出字符为止来解决上述问题。循环缓冲器由三个整型变量: CircIn, CircOut 和 CharsInBuf 来控制。CircIn 指向下一个由中断例程放进输入缓冲的字符, CircOut 指向下一个要从缓冲中取走的字符, CharsInBuf 是在缓冲中等待的字符数。

当没有字符在输入缓冲器中时, CircIn 和 CircOut 相等并且 CharsInBuf 为零。当数据来到串行口, 中断例程将进入的字符加到缓冲器, 并递增 CircIn 和 CharsInBuf, 注意当到达缓冲器尾部时, CircIn 置为 1, 回到缓冲器的始端。这就是缓冲器被称为循环的。

过程 GetCharInBuf 检查 CharsInBuf 是否大于零, 它指出缓冲器中是否有字符存在。如果 CharsInBuf 大于零, 则移去缓冲器中字符并使 CharsInBuf 递减以及 CircOut 递增。数据不断被处理, 所以循环缓冲器绝不应满。AsyncCommunications 程序使用了 1K 缓冲, 这对于大多数通信目的来说都应是足够的。

### § 11.4.3 内存驻留程序(TSR)实现

MS-DOS 被设计成每个时刻只运行一个程序。它不支持多任务, 即同时运行多个程序。这首先是因为单任务操作系统比多任务操作系统容易设计得多, 其次 8088 微处理器速度太慢, 而且没有足够的内存地址以有效地支持多任务。

当 Borland 的 SideKick 程序在 1984 年突然出现在软件市场上时, 它似乎就象魔术。任何时间、任何位置, 一次击键就能调出便笺、计算器和其它有用的实用程序。Sidekick 能够这么做的原因是因为 DOS 提供了终止程序但仍让它们驻留内存的例程, 也就是说 SideKick 是一个内存驻留(TSR)程序, 它可以把自己锁在内存, 即使其它程序运行时它也总是在那儿。

几年之后的今天, 内存驻留程序已经相当普通。即便如此, 它们对大多数程序员来说也仍是神秘的事物。本节将解释我们如何用 Turbo Pascal 写出一个功能受限的内存驻留程序。

#### § 11.4.3.1 解决再入问题

内存驻留程序常被称为 TSR, 因为它们终止(Terminate)但保持驻留(Stay Resident)在计

计算机内存。将一个程序锁进内存是容易的，但一旦它在内存就让它做一些有用的事却可以是极在复杂的。大多数难题来自 DOS 是不可再入的。如果不熟悉再入(reentrant)这个术语，不必过于拘泥，这是简单的技术术语，它意味着不能在同一时刻并行执行两个 DOS 服务。也就是说，如果一个 DOS 服务被另一个 DOS 服务中断，则它将无法再工作。

在通常情况下，由于 DOS 在一个时间只执行一个程序，所以不会有任何问题，但是有了 TSR，问题就复杂了，完全有可能在一个程序执行 DOS 服务时，TSR 也执行另一个 DOS 调用，也就是说，一个 DOS 服务会被另一个 DOS 服务中断，其结果是灾难性的，很可能引起系统崩溃。

处理重入问题有两条途径，第一种方法是最简单的：确保我们的 TSR 决不使用任何 DOS 服务。第二种方法更复杂一些：确保我们的 TSR 在另一 DOS 服务激活时决不执行 DOS 服务。本书中将采用第一种方法，在任何时候都避免使用 DOS 服务。当然这样的 TSR 功能是很受限制的。

#### § 11.4.3.2 用 Turbo Pascal 实现 TSR

在 Turbo Pascal 的 DOS 单元中，有一个过程叫做 keep，它结束我们的程序但保持它驻留。Keep 只要一个 Word 参数，它给 DOS 传送一个程序出口码。Keep 的功能是简单地将程序锁在内存中使之“休眠”，因此在进入 Keep 命令之前还有许多事必须做。其中一件事是，我们必须决定 TSR 一旦驻留如何去激活它。我们也必须安装中断处理程序以使 TSR 工作。

要使 TSR 做任何事，都必须触发它。如果我们有 SideKick，触发器就是按下热键(hot-key)，或者说是键盘中断。另一个好的触发器是计算机的内部时钟，它每秒大约中断18次。事实上，键盘和计时器是最常用来激活 TSR 的触发器。选哪个取决于应用，如果我们需要定期检查某些东西，如一个标志的值等，则用计时中断较好。键盘更适合需要响应用户请求的 TSR。这里给出的程序使用计时器和键 两者来建立一个 TSR，它在键盘有一段时间不活动后关闭计算机的屏幕显示。按下一个键则立刻恢复屏幕。屏幕置影程序保护我们的监视器免于“烧毁”，通常计算机长时间打开但不用，有可能发生监视器“烧毁”。

```
{ $M 2000,0,0}
{ $R-,S-,I-,D+,F+,V-B-,N-,L+}
Program ScreenSaver;
Uses
    DOS, CRT;

Const
    TimerIng = $08;          (* Timer interrupt *)
    KbdInt = $09;            (* Keyboard interrupt *)
    TimeLimit : Word = 5460; (* Wait 5 minutes before blanking *)

Var
    Regs      : Registers;

    Cnt       : Word; (* Counts timer ticks *)
```

```

PortNum      : Word; ( * Port used to disable video * )
PortOff      : Word; ( * Value to disable video * )
PortOn       : Word; ( * Value to enable video * )

```

```

OldKbdVec    : Pointer;
OldTimerVec   : Pointer;

```

```

i            : Real;
Code         : Word;

```

```

( * * * * * )

```

```

Procedure STI;

```

```

InLine( $FB );

```

```

( * * * * * )

```

```

Procedure CLI;

```

```

InLine( $FA );

```

```

( * * * * * )

```

```

Procedure CallOldInt( Sub; Pointer );

```

```

Begin

```

```

InLine( $9C / { PUSHF
               $FF / $5E / $06 }; { CALL DWORD PTR [BP+6] }

```

```

End;

```

```

( * * * * * )

```

```

Procedure Keyboard( Flags, CS, IP, AX, BX, CX, DX,
                   SI, DI, DS, ES, BP : Word ); Interrupt;

```

```

Begin

```

```

CallOldInt( OldKbdVec );      ( * Call original interrupt * )

```

```

If ( Cnt >= TimeLimit ) Then ( * Restore screen, if disabled * )

```

```

    Port[PortNum] := PprtON;

```

```

Cnt := 0;      ( * Reset counter * )

```

```

    STI;End;(* * * * *
*)

```

```

Procedure Clock(Flags,CS,IP,AX,BX,CX,DX,
                SI,DI,DS,ES,BP : Word); Interrupt;

```

```

Begin

```

```

CallOldInt(OldTimerVec);    (* Call original interrupt *)

```

```

If (Cnt > TimeLimit) Then    (* If timer limit is reached, *)

```

```

Port[PortNum] := PortOff    (* disable video

```

```

Else

```

```

    Inc(Cnt,1);              (* Otherwise, increment counter *)

```

```

End;

```

```

(* * * * *

```

```

Begin

```

```

(* If user entered a number parameter, *)

```

```

(* compute the delay factor.          *)

```

```

If paramCount = 1 Then

```

```

    Begin

```

```

        Val(ParamStr(1),i,code);

```

```

If (Code = 0) and (i > 0) and (i < 11) Then

```

```

    TimeLimit := Trunc(i * 18.2 * 60);

```

```

End;

```

```

Regs.AH := $0F;    (* Determine the type of video *)

```

```

Intr($10,Regs);    (* adapter in use (Mono or CGA *)

```

```

If Regs.AL = 7 Then    (* Mono adapter *)

```

```

    Begin

```

```

        PortNum := $3B8;

```

```

        PortOff := $21;

```

```

        PortOn := $29;

```

```

    End

```

```

Else    (* Color adapter *)

```

```

    Begin

```

```

        PortNum := $4D8;

```



```

    PortOff := $ 25;
    PortOn := $ 2D;
    End;
    (* Save original interrupts *)
    GetIntVec(KbdInt, OkdJbdVec);
    GetIntVec(TimerInt, OldTimerVec);

    (* Install new interrupts *)
    SetIntVec(TimerInt, @/clock);
    SetIntVec(KbdInt, @Keyboard);

    Cnt := 0;    (* Initialize counter *)
    Keep(0);    (* Terminate and stay resident *)
    End.

```

注意在所列程序的开始处给出了一个编译指示:

```
{ $M 2000,0,0 }
```

这个指示限制栈段为2000字节,并为堆分配空间。我们必须限制 TSR 所用的内存量,特别是堆。如果我们不限制堆,我们的 TSR 将“吞进”全部可用内存而什么也不给其它程序留下。

刚才给出的 TSR 程序依赖于在单色显示和 CGA 视频适配器上发现的一个有用特性。这个特性允许我们关闭和打开视频显示而不影响显示的内容。写一个特定的值到一个口使视频关闭,而写另一个值到这个口可使视频打开。不幸的是这个口的地址和关/开值取决于所用的适配器,参见表11-20。当 TSR 初启时,它必须判定所用的适配器类型以选择合适的口地址和关/开值。

	Monochrome	CGA
Port Address	3B8h	3D8h
Turn Off	21h	25h
Turn On	29h	2Dh

表11-20 视频关/开控制值

一旦安装完毕, TSR 就开始计秒。当流逝的时间超过一个限度, TSR 就匿影屏幕。计时从最后一次击键算起。这样,如果期限为2分钟,则最后一次击键之后2分钟屏幕将失去显示。屏幕匿影后按下任何键都将恢复屏幕显示,并且 TSR 再度开始计时。

这个 TSR 取决于两个中断处理程序—Keyboard 和 Clock。任何时候按下一个键,都会激活 Keyboard。类似地, Clock 处理程序附加到计时器中断(\$08)上,系统时钟每次中断它都执行。注意这两个中断处理程序都由执行原先中断(被替代的中断)开始,这一点是非常重要的,我们的 TSR 决不能影响键盘中断和时钟中断的正常功能。CallOldInt 过程特别被设计来执行中断。

```

Procedure Keyboard(Flags,CS,IP,AX,BX,CX,DX,
                  SI,DI,DS,ES,BP : Word); Interrupt;

Begin
  CallOldInt(OldKbdVec);      (* Call original interrup *)

  If (Cnt >= TimeLimit) Then  (* Restore screen, interrupt *)
    Port[PortNum] := PortOn;
  Cnt := 0;                    (* Reset counter *)
End;

(***** )

Procedure Clock(Flags,CS,IP,AX,BX,CX,DX,
                SI,DI,DS,ES,BP : Word); Interrupt;

Begin
  CallOldInt(OldTimerVec);    (* Call original interrupt *)

  If (Cnt > TimeLimit) Then   (* If time limit is reached, *)
    Port[PortNum] := PortOff  (* disable video *)

  Else
    Inc(Cnt,1);               (* Increment counter *)
  STI;
End;

```

时钟处理程序执行两个任务。第一，检查计数器是否已经超过时间期限，如果超时，处理程序关闭屏幕，如果未超时，增加计数。Keyboard 处理程序要做的刚好相反，如果计数器超过时间期限，则处理程序恢复屏幕。在任何情况下计数器都复位为零。正如我们所看到的，TSR 并不需要很复杂就可以相当有用。

安装 TSR 需要有四个简单的步骤：

1. 为键盘和时钟保存老的中断例程(向量)。
2. 安装我们自己的中断处理程序。
3. 把计数器初始化为零。
4. 调用 Keep 终止并驻留。

```

(* Save original interrupts *)
GetIntVec(KbdInt, OldKbdVec);
GetIntVec(TimerInt, OldTimerVec);

```

```

(* Install new interrupts *)

```

```
SetIntVec(KbdInt, @Keyboard);
```

```
Cnt := 0; (* Initialize counter *)
```

```
Keep(0); (* Terminate and stay resident *)
```

TSR 例子中设置缺省的时间限制是5460, 大约五分钟, 当装入 TSR 时, 用户可以通过在命令行上加上一个数来覆盖缺省值, 这个数指明在匿影屏幕之前要等待的分钟量。例如, 命令

```
SCRSAVE 1.5
```

设置时间间隔为90秒(1.5分钟)。我们可以规定十分钟以内的任何时间期限。

中断处理程序是最有用和最具挑战性的 PC 程序设计领域, 用 Turbo Pascal, 我们就可以相当容易地写出在远程通信和内存驻留程序中用到的中断处理程序。虽然中断的概念对我们来说也许是新的, 但这是一个值得开发的领域, 掌握中断的用法实际上是大师级程序员的标志。

## 第十二章 优化与调试

一个好的程序不但要求能正确工作,还要求尽可能快。优化是在不损害基本功能的前提下,使程序运行的尽可能快。调试是找出程序中存在的问题,使程序能正确工作。

优化程序的目的之一是提高程序的运行速度。但这不是唯一的,优化的目的还包括减少代码量和所占用的内存量。当然,速度是主要的,在优化过程中,主要应注意优化能取得最大效益的代码,不必在无关紧要的代码上花费过多的精力。不应花了很多时间对代码进行优化,但对程序执行并未有太大的改进。

在优化程序时,应考虑两个因素:第一,代码优化后应产生明显的效果。第二,改进的效果应让用户感觉得到。如果用户感觉不到执行速度有改进,优化就没有取得效果。

在对程序进行优化时,有些代码一眼即可看出应如何优化。有时就应仔细检查才可加以优化。加快程序运行速度最明显的方法是将部分代码用汇编语言,用外部过程或嵌入代码形式插入主代码中,当然这比较困难。此外,可以用 Turbo Pascal 编写高效代码,同样可以获得高速度,但这与汇编语言代码比较,编写,调试和维护要容易得多。

要优化程序就要知道代码修改之后,速度是增加了还是减小了,下面的 Timer 单元中的过程 Clockon 和 clockoff 可以用于测定程序执行中用了多少时间:

```
Unit TIMER;  
( * * * * * )  
Interface  
  
( * * * * * )  
Uses DOS;  
  
Procedure ClockOn;  
  
Procedure ClockOn;  
  
( * * * * * )  
  
Var  
    h,m,s,s100 : Word;  
    StartClock ,  
    StopClock : Real;  
  
( * * * * * )
```

```

Procedure ClockOn;
Begin
  GetTime(h,m,s,s100);
  StartClock := (h * 3600) + (m * 60) + s + (s100/100);
End;

```

```

Procedure ClockOff;;
Begin
  GetTime(h,ms,s,s100);
  StopClock := (h * 3600) + (m * 60) + (s100/100);
  WriteLn('Elapsed time = ',(StopClock - Startclock);0;2);
End;

```

```

End.

```

在 Timer 单元中,用 Gettime 过程从系统时钟取得时间。Clockon 取出时间后换算成秒,存入变量 StartCLOCK 中。CLOCKoff 取出时间后,换算成秒,存入 StopCLOCK,然用 StopCLOCK 减去 StartCLOCK,并输出结果。在测定某段程序的执行时间时,在这段程序之前调用 CLOCKOn,在这段程序之后调用 CLOCKoff。可看到程序执行所需的时间。

对于不同的的计算机,同一段程序的执行时间会有所不同。但具体的时间值并不重要。计时程序主要是用来测定产生同一结果的不同程序的相对执行速度,有时,一个小小的改动,会使速度大大提高;也可能对程序作了大量修改,速度却没有多大提高。

## § 12.1 控制结构的优化

优化程序要干的第一件事就是检查控制结构,因为 Turbo Pascal 提供了各种控制结构。不同的控制结构执行起来差别很大。

常用的控制结构是 If-Then 语句和 case 语句。在多分枝比较时,这两种控制结构都可以使用,但效率却不同。对于 If-then 结构也有两种方法;一种是一个一个的比较,即串行比较。一种是嵌套式。优化控制结构就是减少比较的次数。嵌套式 If-then-Else 结构,要比串行结构有效得多。而 case 结构与嵌入式的效率差不多,有些要比 If-The-Else 更有效,下面的程序对这三种结构进行测试。

```

Program BooleanTest;
Uses CRT, TIMER;
Var
  i : Word;
  a : Char;

  ( * * * * * )

```

Procedure BooleanTest1;

Begin

If (i = 1) Then (\* Comparison number 1 \*)

Begin

a := '1';

End;

If (i = 2) Then (\* Comparison number 2 \*)

Begin

a := '2';

End;

if (i = 3) Then (\* Comparison number 3 \*)

Begin

a := '3';

End;

If (i = 4) Then (\* Comparison number 4 \*)

Begin

A := '4';

End;

If (i <> 1) and

(i <> 2) and

(i <> 3) and

(i <> 4) Then (\* Comparison number 5 \*)

Begin

a := 'X';

End;

End;

( \* \* \* \* \* )

Procedure BooleanTest2;

Begin

If (i = 1) Then (\* Comparison number 1 \*)

Begin

a := '1';

End

Else If (i = 2) Then (\* Comparison number 2 \*)

Begin

A := '2';

End

Else If (i = 3) Then ( \* Comparison number 3 \* )

Begin

a := '3';

End;

Else If (i = 4) Then ( \* Comparison number 4 \* )

Begin

a := '4';

End

Else ( \* Comparison number 5 \* )

Begin

a := 'X';

End;

End;

( \* \* \* \* \* )

Procedure BooleanTest3;

Begin

Case 1 of

1 :

Begin

a := '1';

End;

2 :

Begin

a := '2';

End;

3 :

Begin

a := '3';

End;

4 :

Begin

a := '4';

End;

```

Else
  Begin
    a := 'X';
    End;
  End;
End;

```

```

( * * * * *

```

```

Begin
ClrScr;

WriteLn( 'Random values of i... ');
WriteLn;
RandSeed := 0;
ClockOn;
For j := 1 To 30000 do
  Begin
    i := Random(7);
    BooleanTest1;
  End;
ClockOff;
RandSeed := 0;
ClockOn;
For j := 1 To 30000 Do
  Begin
    i := Random(7);
    BooleanTest2;
  End;
ClockOff;

RandSeed := 0;
ClockOn;
For j := 1 To 30000 Do
  Begin
    i := Random(7);
    BooleanTest3;
  End;
ClockOff;

```



```

WriteLn;
WriteLn;
writeLn(1 = 1...);
WriteLn;

i := 1;
ClockOn;
For j := 1 To 30000 do
    BooleanTest1;
ClockOff;

ClockOn;
For j := 1 To 30000 Do
    BooleanTest2;
ClockOff;

ClockOn;
For j := 1 To 30000 Do
    BooleanTest3;
ClockOff;

writeLn;
writeLn;
WriteLn(1 = 5...);

i := 5;
ClockOn;
For j := 1 To 30000 Do
    BooleanTest1;
ClockOff;

ClockOn;
For j := 1 To 30000 Do
    BooleanTest2;
ClockOff;

ClockOn;
For j := 1 to 30000 Do
    BooleanTest3;

```

ClockOff;

WriteLn;

WriteLn;

ReadLn;

End.

在这段程序中 BooleanTest1 使用的是串行 If—Then 比较, BooleanTest2 是嵌入式 If—Then 比较。BooleanTest3 是 case 结构, 首先分别取 3000 个随机数测试每种结构, 即每种结构任取 3000 个随机数, 进行比较, 测定所需时间。然后因定用 1 对三种结构各进行 3000 次比较, 测定内所需时间, 再取 5 对三种结构各进行 3000 次比较, 测定内容时间, 下面是测试的结果。

	随机值	1	5
BooleanTest1	9.00	3.18	3.96
BooleanTest2	7.97	1.98	2.91
BooleanTest3	7.63	2.03	2.53

从中可以看出, If—Then—else 结构与 case 结构的效率差不多, 但都比 If—then 要高。

在 Boolean Test1 中, 对任何 i 值都必须比较 5 次, 因此效率极差。在 BooleanTest2 中, 情况要好得多, 当 i<sub>1</sub>=1 时只需比较一次, 即使在 i 等于或大于 4 时, 也只需比较 4 次。在 Boolean Test3 中使用的是 Case 结构: 它与 BooleanTest2 中的 If—Then—Else 结构的效率差不多。

在控制结构中经常要用到布尔表达式, 例如 If—then, repeat—until 和 While—Do 等。这些布尔表达式将一个或多个变量进行测试比较。这种测试可以采用多种形式, 如链式和集合比较。

集合比较写起来很紧凑, 但执行起来效率很低。例如测试一个字符变量 Ch 是否在集合 ['A', 'B', 'C'] 中, 使用集合个较为

If Ch In ['A', 'B', 'C'] Then...

若用链式比较则为

If (Ch = 'A') Or (Ch = 'b') or (Ch = 'c') Then...

经测试就可发现, 后一种比较要比前一有效得多。若用 Turbo Debugger 追踪这两条语句, 就可看到链式比较需要的代码较少。

## § 12.2 其它优化方法

除了控制结构可以改变程序的执行速度外, 程序的其它部分也可以改变程序的速度。例如算术运算, 文件操作, 字符串操作等等。

算术运算的速度主要取决于变量类型和操作类型。Integer 型量的运算要比 Real 型变量快得多。加减法要比乘除法快。下面的程序测试了 Integer 型和 Real 型的加法运算。将程序稍加

改变,可以测试其它运算

```
Program MathComp;
Uses CRT,TIMER;
Var
    i,
    a, b, c : Integer;
    x, y, z : Real;

Begin
    ClrScr;
    a := 1;
    b := 1;
    ClockOn;
    For i := 1 To 10000 Do
        Begin
            c := a + b;
        End;
    ClockOff;

    x := 1.0;
    y := 1.0;
    clockon;
    For i := 1 To 10000 Do
        Begin
            z := x + y;
        End;
    ClockOff;
    WriteLn;
    write('Press ENTER... ');
    readLn;
End.
```

下面的表中给出整型和实型变量的加减乘除四则算术运算的执行时间。

	整型	实型
加	0.33	2.03
减	0.33	3.19
乘	0.60	4.89

从表中可以看出, Integer 型几乎比 Real 型快十倍。在 Real 型中, 除法所需时间是乘法的两倍多, 是加减法的四倍多, 因此, 在可能的情况下, 应尽量使用 Integer 型。在使用 Real 型时, 应尽可能不使用除法, 用乘法代替。例如 X

4.0用  $X * 0.25$  代替, 这样执行起来要快得多。

当然在程序中要用到许多计算, 有很复杂的公式。在这种程序中, 应把可读性放在第一位。因为在优化复杂公式时, 对于微小的改动, 可能很难发现问题出在哪里。

优化的另一个方面是文件操作。即使是使用硬盘, 文件输入输出也是非常慢的。使用缓冲区可以改进文本文件的操作速度。对不同大小的文本文件缓冲区, 执行速度是不同的, 如果不说明文本缓冲区, Turbo Pascal 将设置一个128字节的缺省缓冲区。一般来讲, 缓冲区越大, 输入输出操作就越快, 但对于 Turbo Pascal 文本文件并不严格如此。下表给出了不同大小的缓冲区的操作时间。

缓冲区	时间
128字节(缺省)	5.38
256字节	5.16
1023字节	6.87
1024字节	2.20
4096字节	1.43

从表中可看到, 128字节的缺省缓冲区与256字节的缓冲区差不多, 速度都相当慢。可是, 1023字节的缓冲区比缺省的128字节的缓冲区还慢。然后只要再加一个字节, 速度马上提高一大节, 只用2.20秒。

1023字节的缓冲区与1024字节的缓冲区为什么会有这么大的差别? 事实上, 磁盘划分为512个字节的扇区。对软磁盘, 1024个字节作为一簇。对硬盘, 2048到8192个字节作为一簇。当缓冲区的大小与磁盘上的簇的大小一致时, 在对磁盘作读写操作时, 工作量要少些, 因此缓冲区设置为1024的倍数效率最高。

字符串是 Turbo Pascal 中常用的一种数据类型, 字符串处理有 Turbo Pascal 提供的标准过程进行。但是字符串操作, 特别是连接操作, 可能非常慢。下面的两个过程都是建立由100个 'A' 组成的字符串

```

Program TestProcs;
Uses CRT, TIMER;
Var
    i : Integer;

```

( \* \* \* \* \* )

Procedure Concat1;

Var

i := Integer;

s := String;

Begin

s := '';

For i := 1 To 10 Do

s := s + 'A';

End;

( \* \* \* \* \* )

Procedure Concat2;

Var

i := Integer;

s : String;

Begin

For i := 1 To 100 Do

s[i] := 'A';

s[0] := Chr(100); (\* Set string length. \*)

End;

( \* \* \* \* \* )

Begin

ClrScr;

ClockOn;

For i := 1 To 100 Do Concat1;

ClockOff;

ClockOn;

For i := 1 To 100 Do concat2;

ClockOff;

WriteLn;

Write( 'Press ENTER... ');

```
ReadLn;
End.
```

在这段程序，concat1过程用的是连接操作符，需要6.32秒，同时，Concat2过程用的是字符串中字符的下标，只需要0.27秒。

同样，用其它方法替代 Copy 命令，也可以加快速度。例如：

```
S2:=Copy(S1,3,5)
用下面的命令替代
Move(S1[3], S2[1], 5);
S2[0]:=Chr(5);
```

即用 Move 命令把 S1 的一部分复制到 S2，再将 S2 的长度设置好，若都迭代10000次，使用 Copy 命令要用2.58秒。同时，用后面的 Move 命令，只需用1.32秒。

### § 12.3 编译指令

在 Turbo Pascal 中，编译指令控制错误检查，影响执行速度的编译指令有三个：控制下标出界检查的 R 指令，控制堆栈错误检查的 S 指令，和控制短路布尔求值的 B 指令。

R 指令

当 R 指令有效时，Turbo pascal 将在对数组元素赋值或访问时，检查下标是否出界。这些附加代码明显增加了程序的执行时间。看一看下面的例子。

```
{ $R+ }
Program DirectiveTest;
Uses CRT, TIMER;
Var
    i, j : Integer;
    a : Array [1..100] of Byte;

Begin
    CluSer;

    Write( 'Range checking... ');
    ClockOn;
    For i := 1 To 1000 Do
        For j := 1 To 100 Do
            a[j] := 1;
        ClockOff;

    WriteLn;
    Write( 'Press ENTER... ');
```

```

ReadLn;
End.

```

这段程序用{\$R+}指令编译时,运行要用9.45秒。若用{\$R-}指令编译,运行只要3.08秒。因此,在程序开发时,使用出界检查。在开发完成这后,应去掉出界检查,以获得最佳性能。

#### S 指令

堆栈检查指令控制编译器是否在调用一个过程前检查堆栈是否有足够的空间用于存放局部变量。堆栈检查很重要,如果堆栈出现错误很难发现。但是堆栈检查对程序的执行速度也有影响。下面程序表明堆栈检查对程序性能的影响。

```

Program DirectiveTest;
Uses CRT, TIMER;
Var
    i : Integer;
    s : String;

    ( * * * * * )

Procedure Proc;
Var
    x : array [1..5000] of Word;
Begin
End;

    ( * * * * * )

Begin
Clrscr;
s := 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA';

Write('Stack checking... ');
ClockOn;
For i := 1 To 10000 Do
Proc;
ClockOff;

WriteLn;
Write('Press ENTER... ');
ReadLn;

```

End.

这段程序反复调用过程 proc, 这个过程使用一很大的局部数组变量。当用{\$S+}指令编译后, 程序执行用0.88秒。用{\$S-}指令编译后, 程序执行只用0.49秒。在程序开发和测试阶段, 堆栈检查是必须的, 在程序完成后, 应关掉堆栈检查指令。

#### B 指令

B 编译指令控制 Turbo Pascal 的两类布尔求值方式: 完全求值和短路求值。在短路求值下, 一旦布尔表达式的值已确定, 即不再继续计算。因此短路求值比完全求值速度更快, 除非有充分理由, 应使用短路求值, 下面的程序测定两种布尔求值方式的执行时间:

```
{ $B+ }
Program DirectiveTest;
Uses CRT, TIMER;
Var
    a, i : Integer;

( * * * * * )

Begin
    CluScr;
    ClockOn;
    a := 1;
    For i := 1 To 10000 Do
        Begin
            If (a = 1) Or (a = 2) Or (a = 3) Then
                Begin
                    End;
                End;
        End;
    ClockOff;

    WriteLn;
    Write( 'Press ENTER... ');
    ReadLn;
End.
```

在这段程序中, 若是短路求值, 当 a 为1时, 则不必计算后面的布尔表达式。这时, 程序运行时间为0.27秒。若用完全布尔求值, 则要计算整个表达式。程序运行要用0.71秒。



## § 12.4 调用与参数

在 Turbo Pascal 中, 调用一个过程, 都要做许多准备工作。因此, 调用过程要占用时间, 只要把被调用的过程直接移到调用过程中, 就可以提高程序的执行速度。下面的程序表明, 调用过程要多用一些时间:

```
Program TestProc3;
Uses CRT, TIMER;
Var
    i, j : Integer;

( * * * * * )

Procedure DemoProc;
Begin
    j := 0;
    j := 0;
    j := 0;
    j := 0;
End;

( * * * * * )

Begin
    CtrSer;

    ClockOn;
    For i := 1 To 30000 Do
        Demoproc;          ( * Procedure call. * )
    ClockOff;

    ClockOn;
    For i := 1 To 30000 Do
        Begin
            j := 0;
            j := 0;
            j := 0;
            j := 0;
        End;
```

```

ClockOff;

WriteLn;
Write( 'Press ENTER... ');
ReadLn;
End.

```

在这段程序中，过程 DemoProc 对整型变量 j 四次赋值 0。主过程中调用 DemoProc 过程 3000 次，需要 2.26 秒。同样，在主程序中也对 j 四次赋值 0，循环 30000 次只需 1.48 秒。

但是不使用过程调用会增加程序所占的内存空间。因为，如果程序中要多次调用同一过程的话，就必须把这段代码重复好几次。同时，不使用过程调用，程序修改比较困难。在修改时，必须把每一处都修改到。使用过程调用还是不使用要综合考虑。

在 Turbo Pascal 中，向过程传递参数时有两种方法：一是传递变量的地址，称为变参，一种是传递参数的值，称为值参。如果值参是一个很大的数组时，比如 4000 个字节，就要把所有 4000 个字节都要传过去。而变参只需传递一个地址，只需 4 个字，下面的程序显示了变参和值参各自所需的时间：

```

Program TestParams1;
Uses CRT, TIMER;
Type
    a : AType;
    i, j : Integer;

( * * * * * )

Procedure a1(Var a : AType);
Begin
End;

( * * * * * )

Procedure a2(a : AType);
Begin
End;

( * * * * * )

Begin
ClrScr;

```

```

ClockOn;
For i := 1 To 1000 Do
    a1(a);
ClockOff;

ClockOn;
For i := 1 To 1000 Do
    a2(a);
ClockOff;

WriteLn;
Write('Press ENTER... ');
ReadLn;
End.

```

在这段程序中有两个过程：a1和a2。a1过程用的是变参，a2过程用的是值参。参数都是2000个整数的数组。在程序中调用a1过程1000次只需0.06秒，同时，调用a2过程1000次要15.32秒。显然，值参要比变参多用很多时间，因此，应尽量使用变参，但是对变参所做的一切修改都是永久性的，从过程退出后，改变依然存在。而对值参则不然，从过程退出后，改变也就消失。

## § 12.5 Turbo Pascal 调试器

早期的调试器常用的是DEBUG.COM。使用这个调试器，一次可以执行一条汇编语言的指令，观察各个寄存器，发现问题，但是DEBUG.COM有一些明显的不足，通过这个调试器只能看到汇编程序，而且变量表现为地址。在调试高级语言编写的程序时，不仅要看到汇编语言的指令，而且要知道每条高级语言语句对应哪些汇编指令，以及用名字显示变量。

现在有许多性能先进的调试工具，例如Turbo Debugger。但所有这些调试器有一个共同的缺点，必须退出Turbo Pascal才能使用。

Turbo Pascal提供的集成调试器具有其它调试器不具备的优点，不必退出集成开发环境就可以使用。除了最隐蔽的软件问题，可以跟踪发现所有问题。而且使用方便，即使从未用过调试器，也很容易使用。

在使用Turbo Pascal集成调试器之前，必须先源程序中使用{\$D+}编译指令，或在Option

Compiler菜单下激活Debug Information选择项，这使Turbo Pascal将源程序与其目标代码相对应的信息存贮起来。程序如果使用了局部变量，还应使用{\$L+}，或在Option

Compiler菜单中的Local symbol选择项激活L编译指令。这时，Turbo Pascal将把局部变量的信息存贮起来。这样，可以通过名字观察它们。

在编译程序之前，必须激活Debug菜单下的Integrated Debugging选择项，这将保留用于

存贮调试信息的 RAM。这将占用一些内存,如果不调试程序,而且程序需要大量 RAM 时,应去掉这一功能。

在调试程序时,将要执行的下一个程序语句为设亮度。在程序结束之前,或退出调试之前,集成调试器将始终保持高亮度执行条。下面介绍调试中的命令。

#### 执行到光标(F4)

这条命令可以在程序中设置断点。首先将光标移到程序中选定的代码行,然后按 F4 键。这时 Turbo Pascal 将执行程序到光标所在行。当不必逐行观察程序,直接到达某点时,这项功能很有用。

执行到光标这项命令几乎不受什么限制。当执行到光标仍依赖于光标的当前位置时,必须设置断点,并且在取消之前始终保持这一设置。在执行这一命令时应注意,如果程序始终执行不到光标所在行时,将永远不能获得控制权。

#### 追踪(F7)

调试器的基本功能是具有单步执行功能,即一次执行一条语句。按 F7 键 将执行跟踪功能。它执行高亮度执行条指示的语句,并把高亮度执行条移到下一条语句。如果高亮度执行条指示的是函数或过程调用,追踪功能将跳到函数或过程中,并从那开始继续执行。如果不想进入过程或函数中,执行整个过程并移到下一行可以使用 1 步进功能。

#### 步进(F8)

这条命令与跟踪命令一样,一次执行一行代码。所不同的是,步进不进入过程或函数中。遇到过程或函数调用时,作为一条语句执行。高亮度执行条移到下一行语句。在调试时,如果不需要调试过程和函数时,可使用这项功能。

#### 运算(CTRL-F4)

按 CTRL-F4 键打开一个运算窗口,该窗口分为三部分:运算,结果和新值。使用这一窗口可以计算表达式的值,也可以用于修改变量的值。在运算部分可以输入变量名,表达式或数值。按 ENTER 键后,在结果部分将显示其结果,利用新值部分,可以修改变量的值。在运算部分输入变量名,敲 ENTER 键后,结果部分显示 变量的内容。进入新值部分后,可以输入改变后的值。就将变量内容改变为新值。

#### 调用堆栈(CTRL-F3)

在大程序中,预设特定断点跟踪函数和过程时,很容易失去跟踪。按下 CTRL-F3,或者由 Debug 菜单中选择 Call Stack,将打开一个窗口,以当前位置按倒序列出所有过程和函数调用。如果过程带参数,调用堆栈窗口可以根据过程的名字显示这些参数的值。使用光标键可以使任何一项为高亮度。如果按 ENTER 键, Turbo Pascal 将进入高亮度显示的过程的源代码处。这样可以看出在什么情况下可以执行到程序的当前位置。

#### 增加监视变量(CTRL-F7)

观察变量的能力是集成调试器的最重要功能之一,程序错误总是与变量的不希望出现的值有关,或者由这种值所表示。在窗口菜单中选择 watch 项,可以打开观察窗口。或者按 CTRL-F7 增加一观察变量或表达式时,将自动打开观察窗口。

按 CTRL-F7 时,出现 Add WatchA 窗口,如果源文件中的光标指向一个符号,这个符号将自动出现在观察表达式域中。如果这个符号是要选择的,按 ENTER 键即可以把它加到观察列表中,此外,输入表达式,符号和其当前值立即出在观察窗口中。在跟踪窗口的同时,可以看到变量是如何改变的。如果变量发生了意料之外的改变,就可以发现问题所在。

如果观察窗口已经存在,可以在其中加上新的变量。按 F6 键激活观察窗口,然后按 INS 键后,弹出 Add Watches 窗口,输入要加上的变量名后,按 ENTER 键即可。

在观察窗口中可以显示许多个变量,也可以用几种不同的方法显示一个变量。

#### 删除与编辑观察变量

在调试程序时,可以把变量加到观察窗口,也可以从观察窗口删除不需要的变量,删除变量的最简单方法是,激活观察窗口,使高亮度条指向要删去的变量,然后按 DEL 键。

在 Watch 菜单中选择 Remove all watches,可以从观察窗口中删除全部观察变量。

#### 设置断点 (CTRL-F8)

断点是在源代码行上加的标志,它告诉 Turbo Pascal 到达这行时停止执行。当对问题的位置有一个好的估计时,使用断点比跟踪更加有效。在一个程序中可以设置 16 个断点。设置方法也很简单,使光标到断点行,按 CTRL-F8 键,或由 Debug 菜单中选择 Toggle breakpoint。

断点设置的行必须含有可执行语句。不能在空白行,解释行,编译指令,数据说明或程序的其它非执行部分设置断点。

断点是开关式的,如果一行已经设置了断点,再设置一次断点将把断点删除。退出调试后,断点也将消失。断点不是 EXE 程序文件的组成部分。

#### 清除全部断点

由断点对活盒中选择 Clear all breakpoint,可以清除程序中的所有断点。

#### 观察下一个断点

如果程序中用了很多断点,有可能找不到一些断点,在断点对活盒中选择 View Next breakpoint 可以确定断点。Turbo Pascal 将找到下一个断点,装上断点附近的源文件,光标位于设置了断点的行。

#### CTRL-Break

按 CTRL-Break 键,可以在程序中的任何一点暂停程序的执行,程序暂停后,执行光条位于要执行的下一条语句,从外部中断程序时,不能确定程序执行到了什么地方。不过,这一技术对于从任何一点跳入跳出程序还是有用的,特别是终止死循环为一个好方法。

上面是 Turbo Pascal 调试器的一些命令及使用方法,下面再讲几个与调试器有关的问题。首先讲一下显示格式。使用观察窗口可以显示任何类型的变量。若不说明,数据按缺省格式显示。缺省格式如下表:

数据类型	缺省显示格式
Byte, Integer, Word	数值标量按十进制方式显示
Real	若可能,按不带指数的十进制方式显示浮点变量
字符	32以后的 ASCII 码显示字符,0到31之间的 ASCII 控制码,显示前面加 # 符号的十进制值。
布尔变量	显示 TRUE 或 FALSE。
指针	显示段和偏移量,用 CSEG 或 DSEG 分别表示代码段或数据段地址按十六进制格式显示。
字符串	显示为用引号括起来的相连的字符,控制码显示为前面加 # 符号的十进制数。

数组	数组内容显示在括号中,用逗号分开.如果是多维数组,使用嵌套的括号。
记录	记录内容显示为用括号围起来的列表,元素之间用逗号 分开。嵌套的记录用嵌套列表显示。

除使用缺省格式外,观察窗口可以使用格式说明,改变数据显示的方式.使用格式说明非常简单.在观察窗口增加变量时,再简单地加上一个逗号并列出想要用的格式说明符.例如,希望用十六进制格式显示一个整型变量*i*,可以输入*i, x*即可.下面是观察窗口的格式说明符:

格式说明符	结 果
\$H,X	显示标量
C	将特殊字符显示为图形值,而不是十进制值。
D	把标量显示为十进制值
Fn	把浮点数显示为 <i>n</i> 个有效数字. <i>n</i> 可以为 2 到18,缺省值 为11。
M	以十六进制方式显示变量的内存转储.加上 D 说明符后,转储 显示为十进制,而 C 或 S 说明符使转储显示为 ASCII 和特殊字符的字符串。
D	把指针显示为段和偏移地址,两部分都为四位十六进制值。
R	显示记录的域名及其值。
S	用 ASCII 和特殊字符显示串,通常特殊字符显示为十进制值。

在调试时应注意的另一个问题是,集成调试器每行只作为一条语句调试.例如, Turbo Pascal 允许把代码写成如下形式:

```
S:=1*z; z:=i DIV Trunc(sqrt(s)); s:=zDIV 0;
```

在一行中有三条语句.如果三条语句中任何一个中有错误,集成调试器不能指出哪一条语句出现了错误.因此最好写成如下形式:

```
s:=1*z;
z:=i Div Trunc(Sqrt(s));
S:=z Div 0;
```

这样,如果有错误,可以确切知道哪条语句出现了错误。

另一方面,如果一段代码可以保证正确,把它们写在一行可以节省调试的时间.因为这样一次可以调试几条语句。

要注意的是,不能因为有了调试器,就不认真对待程序设计,利用编程技巧防止错误比事后用调试器寻找错误要好得多.应尽可能把程序分解为模块,使用参数而不要用全局变量,在装入程序前调试好模块.这些都对减少错误有帮助。

集成调试器需要占用内存。如果程序太大可能没有足够的内存用于调试。有几种方法可以最大限度地利用内存。首先，如果计算机装有扩展内存，可以保证至少有64K 可用，Turbo Pascal 用这一内存存贮编辑文件。

如果给堆栈和堆分配了多余的内存，实际上用不上，可以减少这些数据区的大小，节省的内存空间可用于调试。此外，去掉内存驻留程序，也可以使调试器有更多的工作空间。

在编译程序时，把 S 和 R 指令这类错误检查编译指令关掉，这些指令会增加程序的大小，在观察局部符号的时候，用 { \$L- } 编译指令减少局部符号表的大小。同样，不需要调试的地方用 { \$D- } 编译指令以节省内存。

为节省内存，也可以把程序组织成覆盖。这会使程序放慢，但可以有更多的内存用于调试。如果可以把过程与函数的调试与主程序分开。这会加快调试速度，并节省内存。

最后，可以修改 Turbo.TPL 文件，用 TPUMOVER 程序从 TURBO.TPL 中去掉不用的单元，不过这种方法有些过分，只应作为最后的手段，在其它方法都不行时才用。

虽然集成调试器功能强大，它也有一些局限性。例如，它不能跟踪进入标准 Turbo Pascal 单元，如 CRT, DOS, GRAPH, GRAPH3, PRINTER, SYSTEM, 及 TURBO3 等。虽然这些单元是由 Borland 公司提供的，不需要再调试。

集成调试器也不能跟踪进入下列过程：外部过程，中断过程，全部用嵌入代码写的过程，不是用 { \$D+ } 编译指令编译的过程，没有源代码的过程，及设置为出口过程的过程，总之集成调试器可以解决99% 的问题。如果绝对需要跟踪程序的每一部分，应考虑使用 Turbo Debugger，它比集成调试器更有力。

程序串设计中出现错误是很难避免的，无论怎样小心，也不能避免逻辑错误，跟踪并改正错误即费时又费力。使用 Turbo pascal 内装调试器可以加快程序的生产。

## 第十三章 对象

现在几乎没有几个程序员不知道面向对象的程序设计(OOP)这一术语,面向对象的程序设计是当今软件开发的最热门方法。在5.5版中,Borland 公司把面向对象的程序设计方法引进了 Pascal 中,在6 版中,作了进一步的改进。

如果从来没使用过面向对象的程序设计这种技术也没关系。只要掌握一些新的术语,与已经知道的概念联系起来,就可以很好地学会面向对象的程序设计。要理解面向对象的程序设计不需要重新学习编程。

面向对象的程序设计冲破了数据、代码分离的传统作法,它是一种全新的程序设计方法,增加了一些新的概念。现在 Turbo Pascal 支持对象,但是用不用对象以及在什么程序上使用对象取决于个人喜好,然而对 Turbo Pascal 6,更有助于掌握面向对象的程序设计概念。Turbo Vision 是一种极为有力的工具集,它几乎完全建立在对象之上的。在下一章中将介绍 Turbo Vision。

### § 13.1 对象的概念

对象主要基于三个主要概念,代码与数据的合并;继承;和封装。对象使程序更接近于现实生活中处理问题的方式。在看电视时,我们并不关心电视机是如何工作的,我们只是打开电视机,调好频道。在面向对象的程序设计时,把代码和数据包在对象中。

一般习惯上,要定义数据结构容纳信息,定义过程和函数对信息进行处理,在面向对象的程序设计中,数据与过程合并为对象。对象既包括实体的特性(数据),也包含实体的动作行为(过程),把这些特性与动作行为合并在一起,就有了完成其功能的一切。

为了理解什么是对象,举一个例子进行说明。一架飞机可以用物理术语描述为载客量,产生的推力,牵引系数等等,另一方面从功能上可描述为起飞,爬升和降落,转向和着陆等等。只用物理描述或只用功能描述都不能抓住什么是飞机的本质,必需同时使用两种描述。

在传统的编程方法中,可以把飞机的物理特性定义为如下的数据结构:

Type

Airplane=record

Airspeed:Word;

Attitude:Word;

Flaps:(Up, Down);

End;

再将飞机的行为定义为过程和函数:

Procedure Accelerate;

Begin

{...}

End;



```

Procedure Decelerate;
Begin
{...}
End;
Procedure FlapsUp;
Begin
{...}
End;
Procedure FlapsDown;
Begin
{...}
End.

```

在面向对象的程序设计中,特性(数据)和行为(过程)可以结合为一个单一的实体,即对象,实义为对象的飞机可以为下面的形式:

```

type
  Airplane=Object
    Airspeed ; Word;
    Altitude; Word;
    Flaps : (Up, Down);
    Procedure Init;
    Procedure Accelerate;
    Procedure Decelerate;
    Procedure Ascend;
    Procedure Descend;
    Procedure FlapsUp;
    Procedure FlapsDown;
  End;

```

上面给出的对象中既有数据的说明也有过程的说明。在面向对象的程序设计语言中,过程与函数在对象中说明为“方法”。注意,对象只定义了方法的头,方法的实际代码另外说明。例如:

```

procedure Airplane. Init;
Begin
  Flaps:=down;
  Airspeed:=0;
  altitude:=0;
End;
Procedure Airplane. FlapsUp;
Begin
  Flaps:=Up;
End;

```

方法是用对象名(Airplane)和过程名(FlapsUp)两者共同定义的。与引用记录的域是一样的。事实上,可以把对象看作包含数据域和方法说明的记录。在方法中,数据域 Flaps 的引用不必再指明对象的名字,在过程名中指明 Airplane 的作用与 With 语句对过程体的作用相同,定义了对象之后,可以用对象名来说明变量,例如:

```
Var
```

```
  A:Airplane;
```

在程序中使用如下语句了:

```
With A Do
```

```
  BBegin
```

```
    Init;
```

```
    FlapsUp;
```

```
    Accelerate;
```

```
    Ascend;
```

```
  End.
```

从中可以看到面向对象的程序设计的优点。所有影响对象的行为可以由引用对象本身取得。

至于过程 FlapsUp 使用哪个数据结构,不会引起混乱。对于前面给出的对象 Airplane 的定义,允许直接访问数据域。例如下面语句是完全合法的:

```
A.Flaps:=Up;
```

但是这并不是一种好方法。在面向对象的程序设计中,直接访问对象的域是不必要的。事实上, Turbo Pascal 6 可以不必直接访问对象的域。

## § 13.2 继承

对象都有自己的数据和方法,但是也可以从其它对象继承数据和方法。面向对象的程序设计的继承来源于嵌套记录的概念,对于嵌套记录 Turbo Pascal 的程序员是很熟悉的。看一看下面的记录定义:

```
Type
```

```
  Ages=0..150;
```

```
  PersonInfo=Record
```

```
    LastName:String[30];
```

```
    FirstName:String[20];
```

```
    Age      :Ages;
```

```
  End;
```

```
  Grades = 0..12
```

```
  StudentInfo=Record
```

```
    Person : PersonInfo;
```

```
    Grade:Grades;
```

```
    Teacher:String[30];
```

```
  End;
```

记录 PersonInfo 含有用于说明任何人的域, 第二个记录 StudentInfo 说明了 Person, 它含有 personInfo 记录中的所有域。这种嵌套可以逐步增加记录结构的复杂度。

在面向过程的程序设计中, 可以使用同样的概念, 构造复杂性越来越大的对象, 下列一段程序说明如何使用嵌套对象:

```
Type
  Ages = 0..15,
  Person = Object
    LastName: String[30];
    FirstName: String[20];
    Age: Ages;
    Procedure Init;
    Procedure SetName[NewFirst, Newlast: String];
  End
  Grades = 0..12;
  Student = Object(person)
    Grade: Grades;
    Teacher: String[30];
    Procedure Init;
  End;
```

注意在 student 的说明中含有对 person 对象的引用:

```
Student = Object(person)
```

通过这一说明, Student 继承了 Person 中的一切一数据和方法。这时可以引用数据域 Student.LastName。在面向对象的程序设计中, person 称作祖先类型, Student 称为后代类型。一个对象可有多个祖先类型。person 是 Student 的直接祖先。而 Student 是 person 的直接后代。祖先和后代构成的整个体系称为“对象的继承”。

对象不仅可以继承数据, 而且也继承方法, 在前面的例子中, 对象 Person 中带有方法 setName, 它一般用于对 Person 赋予一个名字。由于 Student 对象是 Person 对象的后代, 它继承了 setName 方法。这样, 对 Student 赋一个名字与对 Person 赋名字是同一种方法, 利用继承, 不必写同一方法支持两个对象。

在 person 和 Student 中都含有名为 Init 的方法。在这种情况下, 后代不继承祖先的同名方法。在这个例子中, 除了对 Person 对象作的初始化外, 可能在要对 Student 作进一步的初始化。这两个 Init 过程可能如下这种样子:

```
Procedure person. Init;
Begin
  LastName := '';
  Age := 0;
End;
Procedure Student. Init;
  Person. Init;
  Teacher := '';
```

```
Grade:=0;
```

```
End;
```

注意,在对 Student 类型对象进行初始化中的第一步是调用其直接祖先 Person 的方法 Init。虽然对继承对象方法名字可以相同,数据域的名字却不能。对象的数据域命名之后,任何后代对象的数据域不能使用同样的名字。

要确定一个对象中的所有域和方法有时很困难,如果继承较复杂时更是如此。必须仔细追踪继承,看看哪些方法被覆盖,哪些没有,这种检查是很重要的。

继承是一个复杂的课题,它与许多分枝有关,在本章的后面将作进一步的详细讨论,虽然,在一开始,这种复杂性使得面向对象的程序设计看起来很难使用,只要多加练习,就会发现,它们是很容易掌握的。

### § 13.3 封装

面向对象的程序设计的首要目的是封装,即建产一个完整实体的对象,封装的规则之一是程序员不需要直接存取对象中的数据域,代之以在对象中定义方法,处理所有要处理的数据。看下面的对象定义:

```
Person=Object
  LastName:String[30];
  FirstName:String[20];
  Age      :Ages;
  Procedure Init;
  Procedure Display;
  Function GetLastName:String;
  Function GetFirstName:String;
  Function GetAge :Ages;
  Procedure SetLastName(NewLastName:String);
  Procedure SetFirstNmae(NewFirstName:String);
  Procedure SetAge(New—Age:Ages);
End;
```

这里对象 person 是前面一节中的 Person 对象的扩展形式,同样含有三个数据域:LastName, FirstName 和 Age。为了访问这三个数据域,这个对象中定义了所有可能的方法,报告或改变域的值。

封装看起来是相当强化的代码并且比简单的域访问要烦锁得多。封装的主要优点是,通过对限制访问方法,可的自由地改变域,而不会造成任何边际效应。如果发现内存空间不够用,可以把 Name 域的大小减少几个字符。如果使用了适当的封装,这种改变不会影响其它代码。

从4.0版开始,单元就已经是 Turbo Pascal 的一部分,使用单元有助于构造封装。使用单元可以把一个大程序分解为一些容易管理的部分。对象和面向对象的程序设计可以使容易使用单元,在单元的接口部分是对象的说明,方法体是实现部分。一个单元中对象的个数是使意的,但通常在一个单元中定义一个对象,下面看一个的单元形式实现的对象 Person:

```
Unit PERSONS;
```

## Interface

### Type

```
Ages = 0..150; (* in years *)
Person == Object
  LastName : String[30];
  FirstName : String[20];
  Age      : Ages;
  Procedure Init;
  Procedure Display;
  Function GetLastName : String;
  Function GetFirstName : String;
  Function GetAge : Ages;
  Procedure SetLastName(NewLastName : String);
  Procedure SetFirstName(NewFirstName : String);
  Procedure SetAge(NewAge : Ages);
  End;
```

### Implementation

```
Procedure person.Init;
Begin
  LastName := '';
  FirstName := '';
  Age := 0;
End;

Procedure Person.Display;
Begin
  Writeln('Person: ', FirstName, ' ', LastName);
  Writeln('Age: ', Age);
End;

Function person.GetLastName : String;
GetLastName := LastName;
End;

Function Person.GetFirstName : String;
Begin
```

```
GetFirstName := FirstName;
```

```
End;
```

```
Function Person.GetAge : Ages;
```

```
Begin
```

```
GetAge := Age;
```

```
End;
```

```
Procedure Person.SetLastName(NewLastName : String);
```

```
Begin
```

```
LastName := NewLastName;
```

```
End;
```

```
Procedure Person.SetFirstName(NewFirstName : String);
```

```
Begin
```

```
FirstName := NewFirstName;
```

```
End;
```

```
Procedure person.SetAge(NewAge : Ages);
```

```
Begin
```

```
Age := NewAge;
```

```
End;
```

```
End.
```

在这个例子中，把所有与对象 Person 的代码放入一单元中。这样减少了干扰。

在 Turbo Pascal 的第6 版中，引进了一个新的保留字：Private。使用这个保留字可以将对象的说明分成两块：公共区和专用区。公共区的数据和方法可以用于其它所有的模块。专用区的数据和方法只能用于同一模块中的子程序。下面看一个简单对象的定义：

```
Type
```

```
Positions = (On, Off);
```

```
LightSwitch = Object;
```

```
Position : Positions;
```

```
Function GetPosition : Positions;
```

```
Procedure TurnOn;
```

```
Procedure TurnOff;
```

```
End
```

由于方法具有对象 LightSWitch 的全部功能，数据域可以放入专用域：

```
Type
```

```
PosiTions = (On, Off);
```

```
LightSWitch = Objcet;
```

```

Function — getPosition : Positions
Procedure turnOn;
Procedure TurnOff;
Private
    Position : Positions;
End;

```

把 Position 放入专用域后，可防止其它模块直接访问这个域。由方法定义在说明对象的同一模块中，它们完全可以访问专用域。

在对象中加上专用域后强化了封装概念。研究下面的例子，它是 Student 对象的完全定义：

```
Unit STUDENTS;
```

```
Interface
```

```
Uses PERSONS;
```

```
Type
```

```
    Grades = 0..12; (* O is K *)
```

```
    Student Object (Person)
```

```
        Procedure Init;
```

```
        Procedure Display;
```

```
        Function GetGrade : Grades;
```

```
        Function GetTeacher : String;
```

```
        Procedure SetTeacher (NewName : String);
```

```
    Private
```

```
        Grade : Grades;
```

```
        Teacher : String[30];
```

```
    End;
```

```
Implementation
```

```
Procedure student.Init; Begin
```

```
    Person.Init;
```

```
    Grade := 0;
```

```
    Teacher := '';
```

```
End;
```

```
Procedure Student.Display;
```

```
Begin
```

```
    Writeln('Student: ', GetLastName, ', ', GetFirstName, ', Age ', Ggrade);
```

```

Writeln('Grade: ', Grade);
Writeln('Teacher: ', Teacher);
End;

```

```

Function Student.GetGrade : Grades;
Begin
GetGrade := Grade;
End;

```

```

Function Student.GetTeacher : String;
Begin
GetTeacher := Teacher;
End;

```

```

Procedure Student.SetGrade(NewGrade : Grades);
Begin
Grade := NewGrade;
End;

```

```

Procedure Student.SetTeacher(NewName : String);
Begin
Teacher := NewName;
End;

```

End.

在这个例子中，方法具有对数据域的全部访问功能，因此，这些域放入专用区。如果以任何方法修改这些域，不必担心会产生意料之外的边际影响。在方法 Student.Display 中，已经假定 Person 对象中定义的域也是在专用区，在输出 Age 时，不能直接使用 Age 域，必须调用 GetAge 方法。这是因为这两个对象分别在两个不同的单元中说明，后代对象不能访问其祖先的域。

在有些情况下，可能希望把方法也放入专用区中。例如，作为对象的一部分，希望维护一个有序表。为此可能要写一个方法 Sort，由于数据已经排好序，不需要其它模块调用 Sort 方法，可以把这个方法也放入专用区。

## § 13.4 静态方法和虚拟方法

面向对象的程序设计中的方法可分为两种：静态方法和虚拟方法。静态方法容易理解，在执行时也与通常的过程和函数几乎完全一样，本章中到目前为止出现的例子都是静态的。这些方法所以称之为静态方法，是用为在调用其方法时，总是调用同样的那几个。调用在编译



时已经界定了的，称为早期封装。静态方法需要的内存较少，执行也比较快。

另一方面，虚拟方法却不同，在调用虚拟方法时，实际调用是在运行时界定的，即调用实行进行时。这称为后期封装。为了理解这些术语，并看清虚拟方法的优点，看一个容易理解的例子。在前面的对象 Person 中增加下面的方法：

```
Procedure Person. HappyBirthday;  
Begin  
Age := Age + 1;  
WriteLn( 'The following record has been Updated: ');  
Display;  
End;
```

这个方法自动修改 Person 的 Age，然后调用方法 Display，显示这个修改。由于对象 Student 是 person 的后代，它继承了这一新的方法，这时，若执行下列程序：

```
Procedure Static—VS—Virtual;  
Uses PERSONS, STUDENTS, CRT;  
Var  
    astudent: Student;  
Begin  
Clrscr;  
with Astudent Do  
Begin  
Init;  
SetLastName( 'Judd' );  
SetFirstName( 'Jessica' );  
SetAge(5);  
SetGrade(1);  
SetTeacher( 'Mr. Spillman' );  
display;  
WriteLn;  
Happybirthday;  
End;  
readLn;  
End;
```

这时程序运行的结果如下：

```
Student: Judd, Jessica; Age 5  
Grade: 1  
Teacher: Mr. Stillman  
The following receorde has been updated;  
Person: Jessica Jundd  
Age: 6
```

注意：在 Age 修改后，程序显示的记录仅是 Person，而不是整个 Student。这是由于使用

静态方法时，方法 HappyBirthday 仍 person 对象中，因此调用 Astudent. Happybirthday 时，Turbo Pascal 将从 Student 对象进到 Person 对象，在那找到这一方法，在执行 HappyBirthday 时，调用一个名为 Display 的方法，在静态方法中，这一调用自动将方法限定在 Person 对象中，所以不是显示 Student，而是显示 person。

解决这个问题的一种方法是在对象 Student 中重新定义 Happybirthdy 方法。但是这和解决方法与面向对象的程序设计的基本原理相抵触。而向对象的程序设计的主要目的是减少多余的代码，因此，解决方法是使用虚拟方法。虚拟方法使用后期封装，以确定运行时的确切执行路径。

在将静态方法变成虚拟方法时要分两步。首先建立一个构造方法，构造使 Turbo Pascal 了解对这个对象使用虚拟方法。建立构造的方法很简单，只要在对象说明和方法定义中用 Constructor 代替 procedure。

第二步也同样简单，只要在对象说明中的方法头中加上保留字 Virtual 即可，下面是对象 person 和 Student 修改后的说明：

Type

Ages = 0..150 ( \* in years \* )

Person = Object

Constructor Init;

Procedure Display; Virtual;

Procedure GetLastName : String;

Function GetFirstName : String;

Function GetAge : Ages;

Procedure SetLastName(NewLastName : String);

Procedure SetFirstName(NewFirstName : String);

Procedure SetAge(NewAge : Ages);

Procedure HappyBirthday;

Private

LastName : String[30];

FirstName : String[20];

Age : Ages;

End;

Grades = 0..12; ( \* 0 is k \* )

Student = Object(Person)

Constructor Init;

Procedure Display; Virtual;

Function GetGrade : Grades;

Function GetTeacher : String;

Procedure SetGrade(NewGrade : Grades);

Procedure SetTeacher(NewName : String);

```

Private
  Grade : Grades;
  Teacher : String[30];
End;

```

这时方法 Display 在两个对象中都是虚拟方法，而方法 Init 对两个对象都是构造。做了这些改变之后，在调用 Astudent. Happybirthday 仍然是调用 person. HappyBirthday。但在 Person. HappyBirthday 调用 Display 时，它将在 Student 中执行，而不是在 person 中。这时，前面程序的结果如下：

```

Student;Judd,Jessica;Age 5
Grade:1
Teacher:Mr. Spillman
the following record has been Updated:
Student;Judd,Jessica;Age 6
Grade:1
Teacher:Mr. Spillman

```

虚拟方法的说明要遵守三条规则：第一在调用任何虚拟方法之前，必须调用对象的构造方法。第二，在一个对象中说明为虚拟的方法，在其后代对象中的同一方法也必须说明为虚拟的。第三，一个虚拟方法一经说明，在任何后代中，其头都不得改变，即不得增加，改变，或减少参数。或在函数与过程之间改变。

支持虚拟方法的主要结构是 VMT—Virtual Methodtable 虚拟方法表。虚拟方法表是指向虚拟方法地址的表。Turbo Pascal 为每类型对象建立一个虚拟方法表，包括它自己的或继承的虚拟方法，通过维护点一对象类型的地址表，Turbo Pascal 可以确定执行的路径，而这在编译时是不可能确定的。

VMT 结构的前面是两个字，第一个字是使用 VMT 的对象的大小，第二个字是第一个字的负值，用于证实 VMT 是否已经作了适当的初始化。编译指令 {\$R+} 可以打开这一证实方法，这时，Turbo Pascal 将测试 VMT 前二个字的和是否为 0。如果不为 0，Turbo Pascal 将产生 210 运行错误。

Turbo pascal 用构造方法初始化 VMT。因此，在调用任何虚拟方法之前，必须调用对象的构造方法，如果 VMT 没有进行适当初始化，不能执行虚拟方法。使用带虚拟方法的对象时，必须在程序的开始处调用其构造方法，以保证对对象作初始化。

VMT 的其它域含有对象的虚拟方法的地址，所有相同对象类型的变量指向相同的 VMT，在前面给出的例子中，给出了对象 person 和 student 的说明，VMT 只含有一个其它项：Display 方法的地址。

在调用一虚拟方法时，Turbo pascal 将调用变量的地址传到虚拟方法的堆栈中，因此在调用 Astudent. Display 时，变量 astudent 的地址传到堆栈中。这个地址称为自参数，并且总是作为最后一项传到堆栈中，使用这一地址，方法从 VMT 中取出变量的地址，并用在 VMT 中找到的地址执行适当的方法。换句话说，变量向方法传递一个指向其自身的指针，方法用这个指针确定 VMT，VMT 告诉方法执行哪个代码。

VMT 域不是对象的一部分，而是具有对象类型的变量的一部分，只有在调用构造方法时，才会在数据段建立对象的 VMT。如果不调用构造方法，就不会建立 VMT 这时调用任何

虚拟方法都会导致程序的失败。

## § 13.5 对象类型的兼容性

对象类型变量的兼容性与其它通常的 Turbo Pascal 变量有所不同。主要差别在于,祖先类型与后代类型兼容,但反之不成立。例如,在前面给出的程序中定义的变量,下列表达式

```
Aperson := Astudent;
```

是合法的,但是

```
Astudent := Aperson;
```

是非法的,将产生编译错误。这是因为, Astudent 通过继承包含了 Aperson 中的一切。但是 aperson 却不一定含有 Astudent 中的一切。前面给出的合法语句把 Astudent 中的所有 Aperson 中有的值赋给 Aperson,其它的域都忽略。同样,接受 Person 对象变量作为变参以过程也可的接受 Astudent 型变量。因此,说明如下的过程:

```
Procedure ChangeValue(Var Anyperson:Person);
```

```
Begin
```

```
{...}
```

```
End;
```

可以如下调用:

```
ChangeValue(Astudent);
```

初看起来,对象类型兼容性似乎并不重要,但是它是面向对象的程序设计的一个极为有力的概念—多态性的较简单的形式。多态性只是一种形象的说法。它指的是一个过程可以接受各种各样的对象类型,既使在编译时根本不知道是什么样的对象,只要参数变量是 Var 型参数类型的后代,过程就可以接受它。可以定义以 Person 为祖先的任意多个对象, ChangeValue 过程不仅可以接受它们,而且可以任意使用它们。

多态性的另一个重要含义是,如果接受对象的过程是在运行时得到关于变量的信息,可以定义相容的对象,不必重新编译包含那个过程的单元。这增强了可扩充性。可以编译接受多态性变量例程的单元,向用户提供不带源代码的编译好的单元。用户可以建立自己的对象,与编译好的过程一起工作。

虽然传递作为 Var 参数的对象提供了多态性和可扩充性,它仍然是有局限性的。例如, ChangeValue 过程中的代码只知道 Person 型对象,因此不能修改 Person 中包含的域之外的任何其它域。当增加不多几个概念,如虚拟方法和动态对象之后,多态性的实际能力才能充分发挥。

## § 13.6 对象的动态分配

对于对象的动态分配,有两种动态对象,一种是含有作为数据域的指针的对象,另一种是作为指向对象指针的变量,还有第三种,即前面两种的组合。只要掌握了前两种,第三种就不用费多大努力就可以掌握。下面主要讨论前两种。

在对象中带有指针域时,构造方法应有一个新的功能,负责分配动态域的内存,看下面的对象定义:

```

Type
Buffertype=Array[1..1000] of Byte;
Buffer=Object
Buff: ^ buffertype;
constructor Init;
Procedure SetB(NewBuff:Buffer Type);
Procedure WriteToFile (Var TheFile;File); Virtual;
Destructor Done;
end;

```

在对象 Buffer 中有一个动态数据域, 所以构造 Init 应分配内存:

```

Constructor Buffer. Init;
Begin
New (Buff);
End;

```

另外对象 Buffer 中有一个虚拟方法, 构造的第二项功能是初始化 VMT。在使用动态数据域时, 也可以使用分解方法。分解方法的使用与构造方法相似, 对于动态分配的使用是关键性的。在分配一个动态对象变量时, 在堆中为这个对象分配空间供其使用。在要释放这个变量时, 用分解方法利用 VMT, 以便正确释放存储区。分解方法的定义方式与构造方法相似, 下面是一段简单的代码, 用于释放内存:

```

Destructor Buffer. Done;
Begin
Dispose (Buff);
End;

```

Turbo Pascal 6 也支持建立对象的指针。下面是一个简单的例子, 用动态变量分配 Person 型对象:

```

Program Dynamic—Objects;
Uses PERSONS;
Var
  ApersonPtr: ^ Person;
Begin
New (ApersonPtr, Init);
apersonPtr^.SetLastName('Lederman');
WriteLn('The Last name is', apersonPtr^.GetLastName);
Dispose(apersonPtr);
ReadLn;
End;

```

与前一种动态对象不同, 这种不要求改变对象定义的单元。在前面的程序中, 对 New 的调用与以前的不同。这是 Turbo Pascal 的 New 和 Dispose 标准过程的传统用法作了扩展。这两个例程现在可以带两个参数: 一个动态变量和一个过程名。如:

```

New (apersonPtr, Init);

```

这一调用与下面两个调用的效果相同:

```
New (apersonPtr);
```

```
Apersonptr ^ . Init;
```

使用这种把两个组合想来的语法, 可以确保执行这两步, 并且其执行顺序正确, 动态释放是动态分配的境像, 因此, dispose 命令也可以带一个对分解方法的调用。这样做与先调用分解方法, 再做传统的 Dispose 是等价的。由于在面向对象的程序设计中, 指针变量的使用是非常普遍的, 所以应该在每一种对象说明中加上指针类型, 例如, 在本章前面讨论的单元中应在 PERSONS 单元中加上 Pperson = ^ person;

类型, 在 STUDENTS 中加上

```
Pstudent = ^ Student;
```

在本章后面的讨论中都假定已经加上了这些类型。

## § 13.7 多态性

前面我们已经看到, 在作用于对象的子程序中的 Var 型参数怎样使用多态性。这只是多态性的简单形式。下面将会看到, 动态对象与虚拟方法结合在一起, 怎样得到更强有力形式的多样性。

Turbo Pascal 的规则之一是, 所有类型的指针是赋值相容的。这样, 可以向一个子程序传送任何变量的指针。如果向一个方法传送一个指向对象 变量的指针, 而这个方法又调用一个虚拟方法, 这样就得到了一个极为有力的多态性。下面详细讨论。

假设要建立一个人员的表, 首先使用单元 PERSONS, 其中有对象 Person 的说明。然后建立一个动态链表对象, 存贮 person 类型变量的指针。下面是 PERSLIST 单元的接口部分:

```
Unit PERLIST;
```

```
Interface
```

```
Uses PERSONS;
```

```
Type
```

```
PPersonNode = ^ PersonNode;
```

```
PersonNode = Record
```

```
    Personptr: PPerson;
```

```
    Next: PPersonNode;
```

```
End;
```

```
PersonList = Object
```

```
    Constructor Init;
```

```
    Procedure AddPerson(ThisPerson: PPerson);
```

```
    Procedure DisplayList;
```

```
Private
```

```
    First: PPersonNode;
```

```
End;
```

在这里有一个记录类型表示每一个节点。节点只含有指向对象的指针。第二个结构是一个对象, 它就是链表。要初始化这个表, 需要一个构造, 构造 Init 非常简单

```

constructor PersonList. Init;
Begin
First := nil;
end;

```

在这个表的尾部加上一个新的对象的方法与第五章中讲过的方法是相同的:

```

Procedure PersonList. AddPerson(ThisPerson : PPerson);
Var
    TempNode;
    NewNode : PPersonNode;
Begin
New(NewNode);
New(NewNode^.PersonPtr, Init);
NewNode^.PersonPtr := ThisPerson;
NewNode^.Next := Nil;
If First = Nil Then
    First := NewNode
Else
    Begin
        TempNode := First;
        While TempNode^.Next <> Nil Do
            TempNode := TempNode^.Next;
        TempNode^.Next := NewNode;
    End;
End.

```

虽然这个方法接受了一指向对象 `person` 的指针,但是它没有为这个对象分配空间,它只是为一个指向这个对象的新指针分配了空间。因此,任何调用这个方法的模块必须先为这个对象分配内存。

实现的最后一部分是显示表项的代码:

```

Procedure PersonList. DisplayList;
Var
    Current : PPersonNode;
Begin
Current := first
While Current <> Nil Do
    Begin
        Current^.PersonPtr^.Display;
        WriteLn;
        Current := current^.next;
    End;
End.

```

End;

End;

这个方法是通过调用 Person.Display 完成显示的。这里的 display 是一个虚拟方法。

下面是一个测试链表程序的程序，这个程序只是对现有代码生成几个简单的名字：

```
Program TestList;
```

```
Uses PERSLIST, PERSONS, CRT;
```

```
Var
```

```
  APerson : PPerson;
```

```
  AList : PersonList;
```

```
  Count : Integer;
```

```
Begin
```

```
  ClrScr;
```

```
  AList.Init;
```

```
  For Count := 1 To 4 Do
```

```
    Begin
```

```
      New(APerson, Init);
```

```
      With APerson ^ Do
```

```
        Begin
```

```
          SetLastName('Last' + Chr(Count+48));
```

```
          SetFirstName('First' + Chr(Count+48));
```

```
          SetAge(Count);
```

```
          AList.AddPerson(APerson);
```

```
        End;
```

```
      End
```

```
  AList.DisplayList;
```

```
  Readln;
```

```
End.
```

如果使用所有支持单元运行这个程序，输出结果如下：

```
Person:First1-Last1
```

```
Age:1
```

```
Person:First2-last2
```

```
Age:2
```

```
Person:First3-Last3
```

```
Age:3
```



Person:First4—Last4

Age:4

这时,如果认真分析这个例子,可以想到几种更简单的方法完成同一工作,现在可以对继承对象用相同的代码,既使在同一个表中使用不同的对象。多态性就是精确地使用同一些方法支持各种各样形式的对象。这样讲可能难于理解。为此再看一个例子,它与前面的程序稍有不同:

Program Polymorphism;

Uses PERSLIST, PERSONS, STUDENTS, CRT;

Var

  AStudent : PStudent;

  APerson : PPerson;

  AList : PersonList;

  Count : Integer;

Begin

  ClrScr;

  AList.Init;

  For Count := 1 To 4 Do

    If Odd(Count) Then

      Begin

        New(APerson, Init);

        With APerson ^ Do

          Begin

            SetLastName('Last' + Chr(Count+48));

            SetFirstName('First' + Chr(Count+48));

            SetAge(/count);

          End;

        AList.Addperson(APerson);

      End

    Else

      Begin

        New(AStudent, Init);

        With AStudent ^ Do

          Begin

            SetLastName('Last' + Chr(Count+48));

            SetFirstName('First' + Chr(Count+48));

            SetAge(Count);

            SetTeacher('Teacher' + Chr(Count+48));

```

        End;
    Alist. AddPerson(Astudent);
    End;
Alist. DisplayLst;
Readln;
End.

```

在这个程序中，对象类型可以是 person 型，也可以是 Student 型。但是链表仅存贮指针，它可以处理不同类型，不同大小的对象。由于 Displaylist 方法调用虚拟方法 display，它可以显示 Person 的任何后代类型的对象。这段程序的输出结果如下：

```

Person: First1 last1
Age: 1
Student: Last2, First2: Age2
Grade :7
Teacher: Teacher2
Person: First3 Last3
Age: 3
student: last4, First4: Age4
Grade:9
Teacher: Teach4

```

## 第十四章 Turbo Vision

Turbo Vision 是 Turbo pascal 6 增加的主要部分。如果要全面介绍，可以写成上千页的一大厚本书。但是这种书只对少数读者有用，他们需要了解实现的每一内含的结构，这些结构中有几百条新的类型，对象，过程和方法。

对于绝大部分读者来讲，只要了解足够的细节就可以了。在下面的介绍中，将 Turbo Vision 分成几个部分。只要学会了这几个部分就足以写出非常复杂的程序。在本章中使用一个完整的例子，因此每节都是前后相连的：首先引进对象类型，然后讲解这个对象中定义的方法，相关子程序和常数，这些方法分为主要方法和其它方法。主要方法经常要用，其它方法为内部的或者很少使用。

Turbo Vision 是 BorLand 公司 Turbo Pascal 6 编写集成开发环境(IDE)使用的内部机制的接口。通过访问这些机制，可以开发出具有集成开发环境外观与感觉的软件。Turbo Vision 的最大好处是它成为可以重复使用软件的一个庞大的集合。例如，需要建立一个菜单时，只要简单的定义一个 TMenuBox 型对象变量，并将菜单项加到这个对象中即可。如果 TMenuBox 不能处理特殊问题，可以建立一个新的对象类型，继承 TMenuBox 的全部功能，并且增加新的功能。

Turbo Vision 极为灵活，因而也非常复杂，有时候在调用 Turbo Vision 的方法和过程时，可能根本不知道为什么要调用它们，只不过是模仿例子。但是在有了经验之后，就会了解细节。不过有的细节永远不必了解。Turbo Vision 完全是建立在继承和多态性概念之上的。

在这一章中自始至终讨论一个应用，因此，这里的每一个例子都有关连，以便理解。这个应用是一程序，在屏幕上建立和使用显示卡片。象所有 Turbo Vision 程序一样，这个应用程序的外观和感觉与 Turbo pascal 环境一样。

开始先给显示卡片输入数据。每一个卡片有一个“边际”问题和一个“边际”答案。为了简单，每个边可以含只有50个字符的一行文本。

每个卡片输入数据后，就可以把这些卡片存入一个磁盘文件。然后就可以随机地或顺序地运行这些卡片。首先显示边际问题，可以给出答案，或者直接进入下一个卡片。通过选择菜单选择项，或状态行命令，或者按预先定义的热键，可以随时保留这个程序。

上面是 Flashcard—App 程序的工作方式。现在可以逐步开发这一程序了。这有助于用 Turbo Vision 建立自己的程序。在第一次处理大程序时，往往不知如何下手，通常是首先建立自己的用户接口，列出应用程序的功能。Turbo Vision 使这项工作很容易。它几乎立刻取得看得见的结果。

建立 Turbo Vision 应用程序的第一步是建立一个 TApplication 型对象的后代。可以先建立一个简单的后代，不增加新的数据域和非覆盖的的虚拟方法。这样得到如下一段程序：

```
Program Just—an—Application;  
Uses App;  
Type  
  TFlashcardApp = Object(TApplication)
```

```

    End;
Var
    FlashCardApp:Tflashcardapp;
Begin
    FlashCardApp.Init;
    FlashcardApp.Run;
    FlashcardApp.Done;
End;

```

在说明了新的对象类型之后,程序建立了这种类型的一个变量,并调用三个继承的方法。在运行这个简单程序时,最上面一行是一行空白它含有菜单条。屏幕的中间部分称为台面。在最底下一行是状态行,含有提示和热键。缺省状态行中含有一项 ~~ALT-X~~ 退出应用程序。

在运行任何 Turbo vision 应用程序时,必须调用三个方法:Init, Run 和 done, Init 方法初始化 Turbo Vision 要求的所有内部结构。也可以用自己的初始化程序覆盖这个方法,但是必须保证调用方法 TApplication.Init。Run 方法在设定的基础上执行应用程序。done 方法清除所有这些。

TApplication 对象小结(APP.TPU):

继承谱系:

tApplication→TProgram→TGroup→Tview→Tobject

主要方法:

Constructor Init;

初始化 Turbo Vision 的内部子系统;每个应用程序都必须调用这个方法。

Destructor Done;Virtual;

清 Turbo Vision 的内部子系统;每个应用程序都应该调用。

主要继承方法:

由 Tprogram 继承的有:

Procedure HandleEvent(Var Event:TEvent); Virtual;

Procedure InitMenuBar;Virtual;

Procedure InitStatusLine;Virtual;

Procedure Run; Virtual;

由 TView 继承的有:

Procedure GetExtent(Var Extent:Trect):

在对应用程序有了轮廓后,需要定义命令。习惯上,这些命令定义为常数,以字母“cm”开头。Turbo Vision 保留从0到99和从256到999的所有常数。其余的100到255和1000到65535之间的值可用作应用程序的命令。绝大部分保留命令由 Turbo Vision 内部处理,作为用户不用关心它们。但是有几个例外,将在后面讨论。

对于显示卡应用程序,方首先定义下列命令:

Const

CmFileOpen=201;

CmFileSave=202;

```

CmEditAdd = 211;
CmEditChange = 212;
CmEditDelete = 213;
CmNextCard = 221;
CmRandomCard = 222;

```

这些命令使用的数值结构影响支持这些命令的菜单结构。除了不得使用保留值外，使用什么值没有特殊的要求。

使用 Turbo Vision 建立的菜单有两种形式：横式和竖式。与之有关的对象类型分为别 TMenuBar 和 TMenuBox。Turbo Vision 建立一个全局变量 MenuBar，用于连接菜单条和主应用程序，为了将菜单项赋值给这个全局变量，必须覆盖虚拟方法 Tapplication.InitMenuBar。下面是显示卡片的应用程序的菜单条的定义：

```

Procedure TFlashCardApp.InitMenuBar

```

```

  Var

```

```

    R : TRect;

```

```

  Begin

```

```

    GetExtent(R);

```

```

    R.B.Y := 1;

```

```

    MenuBar := New (PMenuBar, Init (R, newMenu(

```

```

      NewSubMenu ('~F~ile', hcNoContext, NewMenu (

```

```

       NewItem ('~O~pen', F3, kbF3, cmFileOpen, hcNoContext,

```

```

        NewItem ('~S~ave', F2, kbF2, cmFilesave, hcNoContext,

```

```

        NewItem ('~Q~uit', 'Alt-X', kbAltX, cmQuit, hcNoContext,

```

```

        Nil))))),

```

```

      NewSubMenu ('~E~dit', hcNoContext, NewMenu (

```

```

        NewItem ('~A~dd', "", 0, cmEditAdd, hcNoContext,

```

```

        NewItem ('~C~hange', "", 0, cmEditChange, hcNoContext

```

```

        ~D~elete', "", 0, cmEditDelete, hcNoContext,

```

```

      NewSubMenu ('~G~o', hcNoContext, NewMenu(

```

```

        NewItem ('~R~andom', F8, kbF8, cmRandomCard, hcNoContext,

```

```

        NewItem ('~N~ext', F9, kbF9, cmNextCard, hcNoContext,

```

```

        Nil))))),

```

```

      Nil)))))),

```

```

  End;

```

建立菜单的第一步是定义菜单的维数积，这只需调用应用对象的 GetExtent。矩形的所有维数都是相对于视界区的，所以左上角为(0,0)。由于菜单条只有一行，必须设置底行为1。菜单的维数只是对菜单的主要部分有效，与其任何子菜单无关。全局变量 MenuBar 实际上是一个指向菜单的指针，必须用过程 New 对它赋值。使用这个过程的扩展形式，New 的第二个参

数是调用 TMenuBar.Init, 这是一极为复杂的构造, 它必须处理任何大小的菜单。它要求矩形的维数和菜单项目作为参数。

在定义菜单项目时, 实际上是把这些项目加到一个链表中。函数 NewMenu、NewSubMenu 和 NewItem 可用于定义这样的表。这几个函数要求的参数中的最后一个是指向下一个菜单项(或子菜单项)的指针, 若是最后一项, 则为 Nil。每个菜单项包括五部分: 菜单项的名字, 菜单项的缩写键的名字(如果有的话), 缩写键的键代码, 应执行的命令和有关帮助键。项名包括一个用(~)字符括起来字符。定义的这个字符可以用于访问菜单项。这里相关帮助键全部设定为全局变量 hcNoContext, 这表示没有相关帮助。键代码全部定义为 Drivers 单元中的常数, 以字母“Kb”开头。

TMenuBar 对象小结(MENUS.TPU)

继承谱承:

TMenuBar → TMenuView → Tview → T Object

主要方法:

Constructor Init(Var Bounds:TRect;AMenu:PMenu);

在指定矩形中建立一个菜单条。

有关子程序:

Function NewMenu(Item:PMenuItem):Pmenu;

在菜单条或菜单盒中建立项目。

Function NewSubMenu(name:TMenuStr;AhelpCtx:Word;

SubMenu:PMenu;Next:PMenuItem):PMenuItem;

在菜单条或菜单盒中建立一个子菜单。

Function NewItem(Name;ShortCutName:TMenuStr;

ShortCutReyCode, ShortCtCommand:Word;AHelpCtx:

Word;Next PMenuItem):PMenuItem;

在菜单或子菜单中定义一个项目。

Function NewLine(Next:PMenuItem):PMenuItem;

在菜单盒中增加一个水平行。

相关类型:

TMenu=Record

Items:PMenuItem;

Default:PMenuItem;

定义一菜单;包括其项及哪个为缺省项。

PMenuItem=^ tMenuItem;

TMenuItem=Record

Next:PMenuItem;Name:PString;Command:Word;

Disabled:Boolean;KeyCode,HelpCtx:Word;

Case Integer of

0:(Param:PString);

1:(SubMenu:PMenu);

End

End

定义一个菜单项。

TMenuStr=String[31];

菜单项的最大长度为31个字符,

其它方法:

Procedure Draw; Virtual;

内部调用,以显示菜单条。

Procedure GetItemRect(Item:PMenultem;Var R:TRect);

Virtual;

内部调用,以确定菜单盒中是否有鼠标器。

TMenuBar 对象小节(Menus.TPU)

继承谱系

TMenuBar→TMenuView→TView→TObject

主要方法:

Constructor Init(Var Bounds:TRect;AMenu:PMenu;

AParentMenu:PMenuView);

建立一个菜单盒,通常是子菜单。

有关子程序:

与 TMenuBar 相同。

其它方法:

与 TMenuBar 相同。

Kbxxxx 常数小结(DRIVERS.TPU)

主要值:

KbF1 ... KbF10, KbCtrlF ... KbCtrlF10, KbShiftF1 ... KbShiftF10, KbAltF ...  
KbAltF10KbAlt11...KbAlt10

KbAltA...KbAltZ

KbEnter,kbesc,kbtabs,kbctrlenter

kbpv,Kbdown,Kbloft,KbRight,KbHome,KbEnd,KbPgUp,KbPgDn KbCtrlLeft,KbCtrlRight,KbCtrlHome,KbCtrlEnd,KbCtrlPgUp,KbCtrlPgDn.

KbCtrlDel.

TRect 对象小结(OBJECT.TPU)

继承谱系:

无

主要方法

procedure Assign(XA,YA,XB,YB:Integer);

对矩形的各角赋值。

有关类型:

Tpoint=Object

X,Y:Integer;

End;

定义屏幕上的点。

其它方法：

Function Contains(P:TPoint):Boolean;

确定点是否在矩形中。

Procedure Copy (R:TRect);

将 R 的值赋予对象。

Function Empty:Boolean;

确定矩形中是否可以填充字符。

Function Equals(R:TRect):Boolean;

确定两个矩形是否相等。

procedure Grow (ADX,ADY:Integer);

用增量改变矩形的大小。

Procedure Intersect(R:TRect);

减小矩形的大小，使之与矩形 R 的交。

Procedure Move (ADX,ADY:Integer);

把增量加到矩形的维数上。

Procedure Union (R:tRect);

增大矩形使之与矩形 R 的并

定义状态行与定义菜单相似，下面是定义显示卡片应用程序的状态行的代码：

Procedure TFlashCardApp. InitStatusLine;

Var

R : TRect;

Begin

GetExtent(R);

R.A.Y := R.B.Y - 1;

StatusLine := New (PStatusLine, Init (R,

NewStatusDef (0, \$FFFF,

NewStatusKey ('~Alt-X~ Exit', kbAltX, cmquit,

NewStatusKey ('~F2~ Save', dbF2, cmFileSave,

NewStatuskey ('~F3~ Opne', dkF3, cmFileOpne,

NewStatusKey ('~F8~ Random', dbF8, cmRandomCard,

NewStatusKey ('~F9~ Next', kbF9, cmNextCard,

NewStarusKdy ('~F10~ Menu', kbF10, cmMenu,

Nil))))),

Nil));

End;

与菜单条一样，这个方法首先把一个矩形赋给状态行的维数，不过这次是在屏幕的底



部。然后调用 Init 初始化指针 StatusLine。Init 过程要求状态键项的表。每个状态键包括三部分：串项，热键代码和要执行的命令。串项通常包括键名和命令名，键名通常用(～)字符括起来，在屏幕上它将显示为高亮度色彩。

TstatusLine 对象小节(MENUS.TPU)

继承谱系：

TstatusLine→TView→TObject

主要方法：

Constructor Init(Var Bounds:TRect;ADefs:PstatusDef);

在给定矩中建立状态行。

有关子程序：

Function NewStatusDef (AMin, Amax: Word; Aitems: PstatusItem; ANext: PStatusDef);

PStatusDef;

建立状态行的项。AMin 与 AMax 表示有关帮助的有效值。

Function NewStatusKey(Atext:String;AKeycode:Word;

ACommand:Word;ANext:PStatusItem);PStatusItem;

在状态行中建立单独一个项，包括键名，键代码和命令名。

有关类型：

PstatusDef= ^ TstatusDef;

TStatusDef=Record

Next:PStatusDef;

Min, Max:Word;

Items:PStatusItem;

End;

定义一状态行，包括有关帮助范围。

PStatusItem= ^ TStatusItem;

TStatusItem=Record

Next:PStatusItem; Text:PString; Keycode:Word; Command:Word;

End;

其它方法：

Destructor Done; Virtual;

释放状态行占用的内存；通常为内部调用。

Procedure Draw;virtual;

内部调用，在屏幕上画出状态行。

Function GetPalette;PPalette;Virtual;

可以被覆盖，以改变颜色结构。

Procedure HandleEvent(Var Event:TEvent);Virtual;

内部调用，以处理状态行事务。

Proceddure Hint(AHelpCtx:Word):String;Virtual;

必须被覆盖，以在状态行提供一行帮助信息。

Constructor Load(Var S: TStream);

内部调用，从一个流中装载一个状态行。

```
Procedure Store (Var S:TStream);
```

内部调用，把状态行存贮到一个流中。/Procedure Update;

在有关帮助之上重新显示状态行。

每个应用对象的外层都必须有一个现场处理器。可以通过覆盖方法 TProgram. HandleEvent 来建立现场处理器。任何现场处理器必须首先调用其祖先的现场处理器。这一结构的其余部分可以很容易从例子中复制。下面是显示卡片的现场处理器：

```
Procedure TFlashcardApp. HandleEvent (Var Event; TEvent);
```

```
Begin
```

```
TApplication. HandleEvent (Event);
```

```
If Event. What = evCom and Then
```

```
Begin
```

```
Case Event. Command Of
```

```
cmFileOpen      ; OpenFile;
```

```
cmFileSave      ; SaveFile;
```

```
cmEditAdd       ; AddCard;
```

```
cmEditCahnge    ; CahngeCard;
```

```
cmEditDelete    ; DeleteCard;
```

```
cmRandomCard    ; RandomCard;
```

```
Else
```

```
Exit;
```

```
End;
```

```
ClearEvent (Event);
```

```
End;
```

```
End;
```

所有的现场处理器在 If 语句中嵌套同样的 Case 试句。Case 语句的选择项是应用程序的命令。当现场处理器接收到命令后，就调用处理这个命令的过程。为了说明，把所有每个命令处理过程当作空结构对待。下面是显示卡片命令的结构：

```
Procedure AddCard;
```

```
Begin
```

```
End;
```

```
Procedure SaveFile;
```

```
Begin
```

```
End;
```

Procedure OpenFile;

Begin

End;

Procedure ChangeCard;

Begin

End;

Procedure DeleteCard;

Begin

End;

Procedure NextCard

Begin

End

Procedure RandomCard

Begin

End;

使用上面刚定义的三个方法，可以建立 TFlashcardApp 对象的最后定义：

type;

TFlashCardApp = Object(TApplication)

Procedure InitMenuBar; Virtual;

Procedure InitStatusLine; Virtual;

Procedure HandleEvent(Var Event; TEvent); Virtual;

几乎每个应用都将覆盖这三个方法，也可以选择覆盖 TApplication. Init 方法，其中包括程序的初始化代码，不过通常使它们分开。

Tprogram 对象小结(APP, TPU)

继承谱系：

TProgram → TGroup → TView → TObject

主要方法：

Procedure HandleEvent(Var Event; TEvent); Virtual;

在每个应用程序中允许为现场处理面覆盖。覆盖它的任何方法在其第一条语句中调用这

个方法、如果未覆盖，它只能处理内部现场。

Procedure InitMenubar; Virtual;

覆盖后在屏幕顶部建立一个菜单条。如果未覆盖，它将建立一空菜单条。

Procedure InitStatusLine; Virtual;

覆盖后在屏幕底部建立一个状态行，若未覆盖，建立的状态行将只有 ALT-X quit 命令。

procedure Run; Virtual;

内部调用，启动应用程序的现场处理器。

有关的全局变量：

application: Papplication = Nil;

存贮主应用对象的指针。

DeskTop: PDesktop = Nil;

存贮应用程序的主菜单(条或盒)的一个指针。

StatusLine: PStatusLine = Nil;

存贮应用程序状态行的指针。

有关类型：

TEvent = Record

What: Word;

case word of

{...}

evMessage: (Command: Word {...});

End;

定义现场，内部信息已省略。

其它方法：

Destructor Done;

内部调用，处理平台，主菜单和状态行。

Procedure GetEvent (Var Event: TEvent); Virtual;

内部调用，检查一个现场是否已经出现。

Procedure GetPalette: PPalette; Virtual;

可以被覆盖，改变整个应用程序的色彩结构。

Procedure Idle: Virtual;

可以被覆盖，在现场未出现时，建立背景模式。

Constructor Init;

内部调用，初始化内部变量。

Procedure InitScreen; Virtual;

内部调用，改变屏幕模式。

Procedure OutOfMemery; Virtual;

可以覆盖，处理程序内存不足的情况。作为缺省，它什么也不做。

Procedure PutEvent (Var Event: TEvent); Virtual;

内部调用，把当前现场存贮在内部缓冲区。

Procedure SetScreenMode(Mode: Word);

设置屏幕模式。其值有 SmCo80, SmBW80, 和 smMone. 也可以包括扩展行模式的 sm-Font8×8。

Function ValidView(P: Pview): Pview;

内部调用, 检查视图区的有效性, 并保证有足够的空间建立它。

TDeskTop 对象小结(APP.TPU)

继承谱系:

Tdesktop→TGroup→Tview→TObject

主要方法

Procedure Cascade(Var R: TRect);

把所有窗口按级分层排列, 这通常不包括对话框。

Procedure Tile(Var R: TRect);

分层排列所有的窗口, 通常不包括对话框。

主要继承方法:

从 TGroup 继承的方法有(见 TRroup 小节):

procedure Delete(P: PView);

Function ExecView(P: PView): Word;

Procedure Inset (P: PView);

其它方法:

Procedure HandleEvent(Var Event: TEvent), Virtual;

内部调用, 处理平台现场。

Constructor Init(Var Bounds: TRect);

内部调用, 建立平台。

Function NewBackground; PView; Virtual;

可以覆盖, 改变缺省背景。

Procedure TileError; Virtual;

在调用 tile 出现错误时, 内部调用。缺省时, 什么也不做。

TGroup 对象小结(VIEWS.TPU)

继承谱系:

TGroup→TView→TObject

主要方法:

procedure Delete(P: Pview);

从一组中去掉一个显示区。

Function Execview(P: Pview): Word;

执行一个标准显示区, 控制停留在这个显示中, 直到接到适当的退出显示区命令。退出命令由此函数返回。

procedure Insert(P: Pview);

将一个显示区增加到一个组中。这通常使显示区出现在屏幕上。

其它方法:

对象组中的对象几乎都未显式建立, 所以它们的内部方法没有什么价值。具体方法看每

个后代对象类型。

虽然前面的方法可以编译执行，但是对支持开发程序没有什么用途。下面介绍一种更有效的方法，它用于显示已开发代码的特定部分。这要用到 Turbo Vision 的对话框。熟悉 Turbo Pascal 环境的用户已知道如何操作对话框，以及哪种输入

输出类型可以表示。建立自己的对话框实际上就是初始化一个空盒，然后插入需要的部分。

为了便于理解对话框设计的方法。看一个简单的例子，建立一个非常简单的对话框，其中只有一个键。当这个对话框显示在屏幕上时，用户必须按那个键，使程序继续，下面是建立这个简单对话框的过程：

```
Procedure SimpleBox(Name : String);

Var
    Dialog : PDialog;
    R : TRect;
    Control : Word;

Begin
    R.Assign(20, 9, 55, 15);
    Dialog := New(PDialog, Init(R, Nname));
    With Dialog ^ Do
        Begin
            R.Assign(4, 2, 29, 4);
            Insert(New(PButton, Init(R, '(not implemented yet)'
                cmOK, bfDefault)));
        End;
    control := DeskTop ^ .ExecView(Dialog);
End;
```

建立对话框的第一步是在屏幕上分配一个区域，对话框将在这个区域显示。如菜单和状态行的完全一样，用 R.Assign 为对话框分配区域。然后，用 New 的对话框指针分配内存。这样建立一个空白对话框，带有由参数提供的标题。

第二步是加上一个或几个标准对象。在这个程序中是选择一个按键。在为这个按键分配一个矩形区域后，调用 Insert 初始化这个按键。这一初始化过程包括将要显示的文本和按这个键时将产生的现场。

对话框定义好后，可以调用 ExecView。这个函数显示对话框，并且只接受对话框定义的命令，而不接受其它命令。这使得这个对话框成为一个标准显示区。现在可以在前面给出的空结构程序中加上对 SimpleBox 的调用，这个过程处理命令。下面是对前面给出的程序的改进版的几个例子；在其中增加了调用 Simple Box：

**Procedure AddCard;**

**Begin**

**SimpleBox( 'Add Card' );**

**End;**

**Procedure ChangeCard;**

**Begin**

**SimpleBox( 'Change Card' );**

**End;**

**Procedure SaveFile;**

**Begin**

**SimpleBox( 'Save File' );**

**End;**

**Procedure OpenFile;**

**Begin**

**SimpleBox( 'Open File' );**

**End;**

**TDialog 对象小结(DIALOGS.TPU)**

**继承谱系:**

**TDialog → TWindow → TGroup → TView → TObject**

**主要方法:**

**Constructor Init(Var Bounds:TRect;ATitle:TTitlestr);**

**用指定维数和标题建立对话框。**

**有关常数:**

**cmOK=10;**

**当用户按 OK 键或 ENTER 时设置现场。**

**cmCancel=11;**

**当用户按 Cancel 键或按 ESCAPE 时设置现场。**

**cmYes=12;**

**当用户按 Yes 键时设置现场, 如果有这个键的话。**

**cmNo=13;**

**当用户按 No 键时设置现场, 如果有这个键的话。**

主要继承方法:

由 TGroup 继承的有:

procedure Insert(P: PView);

由 TView 继承的有:

Procedure GetData(Var Rec); Virtual;

Procedure SetData (Var Rec); Virtual;

其它方法:

procedure HandleEvent(Var Event: TEvent); Virtual;

内部调用, 处理 cmCK, cmCancel, cmYes, 和 cmNo。

其它所有现场由其祖先对象产生。

Function GetPalette: PPalette; Virtual;

可以覆盖, 改变对话框的色彩。

Function Valid (Command :Word); Boolean; Virtual;

内部调用, 确定命令是否有效。

TView 对象小结(VIEWS.TPU)

继承谱系:

TView→TObject

主要方法:

Procedure GetData(Var Rec); Virtual;

根据参数的大小, 从其内部结构向参数复制一些字节。

Procedure GetExtent(Var Extent: TRect);

返回一个矩形, 以(0,0)为原点, 与显示区的当前大小匹配。

Procedure SetData(Var Rec); Virtual;

根据参数的大小, 由参数向其内部结构复制一些字节。

其它方法:

显示对象尚未显式建立, 所以其内部方法没有什么价值。具体方法见其每个后代对象的类型。

TObject 对象小结(OBJECTS.TPU)

继承谱系:

无

主要方法:

Constructor Init;

为对象分配空间, 并初始化所有的域, 即设置为0。

Procedure Free;

清除一个对象。

其它方法:

Destructor Done1; Virtual;

由 Free 内部调用, 处理从对象收回内存。

前面介绍了如何开发一个应用程序的原型, 它可以支持任何 Turbo Vision 程序, 在其中只介绍了设计方法和接口。下面开始在程序中加上实际内容、首先介绍一个最低水平的对象,



显示卡片应用程序，它只表示一个单一的显示卡片。在下面的设计中主要围绕两个对象：TFlashCard 和 TFlashCardSet。TFlashCard 建立一个单独的显示卡片，它的实现很简单。与上一章中的对象 Tperson 的形式相同。看一下 FLASHCRD 单元即可知道：

```
Unit FLASHCRD;
```

```
Interface
```

```
Uses Objects;
```

```
Const
```

```
  QAMaxLen = 50;Type
```

```
  Flashstr = String[QAMaxLen];
```

```
  PFlashCard = ^ TFlashCard;  TFlashCARD = Object(TObject)
```

```
    Constructor Init;
```

```
    Function GetNumber : Integer;
```

```
    Function GetQuestion : FlashStr;
```

```
    Function GetAnswer : FlashStr;
```

```
    Procedure SetNumber(NewNumber : Integer);
```

```
    Procedure SetQuestion(NewQuestion : FlashStr);
```

```
    Procedure SetAnswer(NewAnswer : FlashStr);
```

```
Private
```

```
  Number : Integer;
```

```
  Question : FlashStr;
```

```
  Answer : FlashStr;
```

```
End;
```

```
Implementation
```

```
Constructor TFlashCARD. Init;
```

```
Begin
```

```
  Question := '';
```

```
  Answer := '';
```

```
End;
```

```
( * * * * * )
```

```
Function TFlashCard. GetQuestion : Integer;
```

```

Begin
GetQuestion := Question;
End;

```

```

( * * * * * )

```

```

Function TFlashCard.GetQuestion : FlashStr

```

```

Begin
GetQuestion := Question;
End;

```

```

( * * * * * )

```

```

Function TflashCard.GetAnswer : FlashStr;

```

```

Begin
GetAnswer := Answer;
End;

```

```

( * * * * * )

```

```

Procedure TflashCard.SetNumber(NewNumber : Integer);

```

```

Begin
Number := NewNumber;
end;

```

```

( * * * * * )

```

```

Procedure TFlashCard.SetQuestion(NewQuestion : FlashStr);

```

```

Begin
Question := NewQuestion;
End;

```

```

( * * * * * )

```

```

Procedure TflashCard.SetAnswer(NewAnswer : FlashStr);

```

```

BEGin

```

```

Answer := NewAnswer;
End;

```

```

End.

```

在这个单元中实现的 TFlashCard 充分体现了封装的概念：其方法包括了所有可能的功能，无须访问数据域。对于 TFlashCardSet，它表示一组显示卡片。

Turbo Vision 中有一个称作 Tcollection 的对象类型。同时，它有几个后代对象类型。这些集合对象独立于用户的接口软件，所以很容易用于任何应用程序。这些集合一个最大优点是多态性：使用适当，可以把不同的对象存贮在同一个集合中，如果希望集合始终进行分类，应使用 TsortedCollection。不过应注意，分类集合不支持重复项。如果需要支持重复项，必须建产自己的后代对象。

Turbo Vision 的集合是一个指针表。因此，使用集合对象不需要建立后代对象类型。但是为了适当封装，需要经常建立后代对象，改进对象类型的功能。为了讲得更清楚，考察一下 FLASHSET 单元中的 TFlashCardSet 对象的定义：

```

Unit FLASHSET;

Interface

Uses Objects, FLASHCRD;

Type
  PFlashCardSet = ^ TFlashCardSet;
  TFlashdCardSet = Object(TCollection)
    Constructor Init;
    Function NewCard : PFlashCard;
    Procedure Replace(ACard : PFlashCard);
    Procedure Deleta(ACard : PFlashCard);
    Function NextCard : PFlashCard;
    Function AnyCard : PFlashCard;
  Private
    CurrentCard : Integer;
  End;

```

这个对象的定义提供了显示卡片集的全部功能。初始化主要是通过调用 TCollection. Init 进行的，例如：

```

Constructor TFlashCardSet. Init;
Begin
  TCollection. Init(10,5);
  CurrentCard := -1;
  Randomize;

```

end;

Init 要求两个参数：集合的初始大小和超过初始大小后，集合增加的量。当集合变大后，集合的增长要花费相当长的时间。调用 Randomize 是为了支持随机选择显示卡片的能力。

通常在集合中增加一个对象的方法是，输入数据，然后把指针加到集合中。但是从面向对象的意义上讲，显示卡片方法的工作方法更象这样：“给出一个空白卡片，在两面填上数据，放到集合中”。下面是 NewCard 方法：

```
Function TFlashCardSet.NewCARd:PFlashCard;  
Var  
    BlankCard:PFlashCard;  
Begin  
    New (BlankCard, Init);  
    TCollection.Insert (BlankCard),  
    BlankCard^.SetNumber (Count);  
    NewCard := BlankCard;  
End;
```

象大多数计算机程序一样，集合的项是从0到N-1。但是在显示卡片中，计数是从1到N。这种计数法在 Replace 和 Deete 方法中是明显的，如：

```
Procedure TFlashCardSet.Replace (ACard:PFlashCard);  
Begin  
    TCollection.AtPut (ACard^.GetNumber-1, ACard);  
End
```

以及

```
Procedure TFlashCardSet.Delete (ACard:PFlashCard);  
Begin  
    TCollection.AtDelete (ACard^.GetNumber-1);  
End;
```

下面的两个方法顺序或随机地从一个集中检出卡片：

```
Function TFlashCardSet.NextCard : PFlashCard;  
  
Var    ACard : PFlasChard;  
  
Begin  
    CurrentCard := CurrentCard + 1;  
    If CurrentCard = Count Then  
        CurrentCard := 0;  
    ACard := TCollection.At (CurrentCard);  
    ACard^.SetNumber (CurrentCard+1);  
    NextCard := ACard;  
End;
```

( \* \* \* \* \* )

Function TFlashCardSet.AnyCard : PFlashCard;

Var

ACard : PFlashCard;

Begin

CurrentCard := Random(Count);

ACard := TCollection.At(CurrentCard);

ACard^.SetNumber(CurrentCard + 1);

AnyCard := ACard;

End;

这两个方法都是设置显示卡片的号数。虽然这个数字是存贮在集合对象时，不知道对象的顺序，所以不会修改任何数据。因此，显示卡片号存贮在集合中不是必不可少的。

TCollection 对象小结(OBJECTS.TPU)

继承谱系:

Tcollection → TObject

主要域:

Count: Integer;

确定集合当前大小的唯一方法是访问这个数据域。

主要方法。

Constructor Init(ALimit, ADelta: Integer);

使用要求的初始大小建立一个集合对象。当超过这个大小时，集合将以 ADelta 增加。

Function At(Index: Integer): Pointer;

得到集合中第 n 项的指针。

procedure AtDelete(Index: Integer);

删除集合中的第几项。

Function AtInsert(Index: Integer; Item: Pointer);

在第 n 位增加一个新的项。

Index 项之后的项位置都增加1。

Function AtPut(Index: Integer; Item: Pointer);

替换集合中的第 n 项。

Procedure Delete(Item: Pointer);

删除集合中的指定项。

Procedure DeleteAll,

删除集合中的所有项。

Destructor Done; Virtual;

释放整个集合。

Function Firstthat(test: Pointer): Pointer;

寻找集合中的指定项。Test 是确定条件的布尔函数的地址。这个函数对调用它的子程序是局部的，并且必须说明为 Far 函数。

Procedure Foreach(Action: Pointer);

对集合中的每一项执行特定的过程。Action 是对单个一项执行的过程的地址。这个过程对调用给定的子程序必须是局部的，并且必须说明为 Far 函数。

Function IndexOf(Item: Pointer): Integer; Virtual;

确定集合中项的索引。

Procedure Insert(Item: Pointer): Virtual;

在集合的尾部增加一项。

Function Lastthat(Test: Pointer): Pointer;

在集合中逆序寻找指定项。Test 是确定条件的布尔函数的地址。对调用这个函数的子程序，它必须是局部的，并且必须说明为 Far 函数。

其它方法：

Procedure Error(Code, Info: Integer); Virtual;

内部调用，处理错误。缺省时，这个过程引起一个运行时错误。

Function GetItem(Var S: TStream): Pointer; Virtual;

内部调用，从一个流中恢复一项。如果项的类型不是 TObject 的后代，必须被覆盖。

Constructor Load(Var S: TStream);

内部调用，从一个流中装载集合。

Procedure Pack;

删除所有指向 Nil 的项。

Procedure PutItem(Var S: TStream; Item: Pointer): Virtual;

内部调用，把一个项写到流中，如果项的类型不是 TObject 的后代，必须被覆盖。

Procedure SetLimit(ALimit: Integer): Virtual;

改变集合的大小。虽然集合会自动增大。在删除很多项后，可能有必要缩小集合。这个过程实际上是把所有项拷贝到一个较小的集合中，因此，要用很多时间。

Procedure Store(Var S: TStream);

内部调用，把一个集合写到一个流中。

TSortedCollection 对象小结(OBJECT.TPU)

继承谱系：

TSortedCollection → TCollection → TObject

主要方法：

Function Compare(Key1, Key2: Pointer): Integer; Virtual;

必须用比较两个关键字的函数覆盖，根据比较返回 -1, 0, 或 1。

Function IndexOf(Item: Pointer): Integer; Virtual;

寻找项并返回其索引。如果项不在集合中，返回值为 -1。

Procedure Insert(Item: Pointer): Virtual;

在集合中适当位置增加一项。

主要继承方法：

从 TCollection 继承的有:

Count: Integer;

Constructor Init(ALimit, ADelta);

Function FirstThat(Test: Pointer): Pointer;

Procedure ForEach(Action: Pointer);

Function LastThat(Test: Pointer): Pointer;

其它方法:

Function KeyOf(Item: Pointer): Pointer; Virtual;

如果 Item 不是全部关键字, 这个函数可以被覆盖。

Function Search (Key: Pointer; Var Index: Integer): Boolean; Virtual;

内部调用, 在集合中寻找关键字。

TStringCollection 对象小结(OBJECT.TPU)

继承谱系:

TStringCollection → TSortedCollection → TCollection → TObject

主要继承方法:

从 TSortedCollection 中继承的有:

Function Index = f(Item: Pointer): Integer; Virtual;

Procedure Insert(Item: Pointer); Virtual;

从 TCollection 中继承的有:

Count: Integer;

Constructor Init(ALimit, ADelta);

Function FirstThat(Test: Pointer): Pointer;

Procedure ForEach(Action: Pointer);

Function LastThat(Test: Pointer);

其它方法:

Function Compare(Key1, Key2: Pointer): Integer; Virtual;

纯 ASCII 比较; 可以覆盖, 用于不同比较法的分类。

Procedure FreeItem(Item: Pointer); Virtual

内部 用, 释放集合中的一个串。

Function GetItem(Var S: TStream): Pointer; Virtual;

内部调用, 从一个流中读一个串。

Procedure PutItem(Var S: TStream; Item: Pointer); Virtual;

内部调用, 把一串写到流中。

前面介绍了显示卡片应用程序的用户接口的轮廓和内部机制。下面将把所有这些结合到一起, 建立一个工作程序。其中大部分命令都涉及到对话框的使用。下面举例说明对话框。首先介绍 AddCard 过程:

Procedure AddCard(Var To/Set : TFlashCardSet);

Type

```

TEditData = Record
  Ans : FlashStr;
  Quest : FlashStr;
End;

Var
  Dialog : PDialog;
  R : TRect;
  Control : Word;
  Q, A : PView;
  CardNumber : String;
  EditData : TEditData;
  ACard : PFlashCard;

Begin
  ( * repeat as long as the user wants to make additions * )
  Repeat

    ( * create the dialog box * )
    R.Assign(20, 6, 55, 20);
    Dialog := New(PDialog, Init(R, 'Add Card'));
    With Dialog ^ Do
      Begin

        ( * insert the card number as static text * )
        New (ACard, Init);
        ACard := ToSet.NewCard;
        Str(ACard ^ .GetNumber, CardNumber);
        R.Assign(3, 2, 29, 3);
        Insert(New(PStaticText, Init(R, 'Card Number : '+ CardNumber)));

        ( * insert the answer InputLine with a label * )
        R.Assign(2, 8, 29, 9);
        Insert(A);
        Insert(New(PInputLine, Init(R, QAMaxLen)));
        Insert(A);
        R.Assign(2, 7, 12, 8);
        Insert(New(PLabel, Init(R, 'Answer:', A)));

        ( * inset the OK command as a button * )

```



```

R.Assign(5, 11, 15, 13);
Inset(New(PButton, Init(R, '~O~K', cmOK, bfDefault)));

( * insert the Cance lcommand as a button * )
R.Assing(18, 11, 28, 13);
Inset(New(PButton, Init(R, 'Cancel', cmCancel, bfNormal)));

( * insert the question InputLine with a label * )

R.Assign(2, 5, 29, 6);

Q := New(PInputLine, Init(R, QAMaxLen));
Inset(Q);
R.Assign(2, 4, 12, 5);
Inset(New(PLabel, Init(R, 'Question:', Q)));

( * give values to the input lines * )
EditData.Quest := ACard^.GetQuestion;
EditData.Ans := ACard^.GetAnswer;
SetData(EditData);
End;

( * execute the dialog box * )
Control := DeskTop^.ExecView(Dialog);

If Control <> cmCancel Then
  Begin

    ( * save the information entered * )
    Dialog^.GetData(EditData);
    ACard^.SetQuestion(EditData.Quest);
    ACard^.SetAnswer(EditData.Ans);
    ToSet.Replace(ACard);
  End
Else

  ( * ignore the information entered and remove the card * )
  ToSet.delete(ACard);
Until Control = cmCancel;

```

End;

这个过程首先建立一个对话框,然后在这个盒中加上七个对象。第一个对象是卡片号。由于用户不能改变这个号码,应该使用 PstaticText 对象。后面的两个对象实际上是在对话框的上部,最后加上它们是因为它们是缺省项。这几个对象由 tInputLine 和 TLabal 组成。TInputLine 用入输入数据, TLabel 是输入行的标号。这个标号与输入行对象链接在一起,当用户选择这个项时,标号将为高亮度。其它对象是另外一对输入行和标号及两个键。

在定义对话框后,如果需要数据项,就必须对数据区赋初值。这个处理需要用到 TeditData 的定义。在对这个记录的域赋值之后,调用方法 SetData,把数据存到对话框的内部结构中。

在数据值设定之后,调用 ExecView 方法,这个方法用于完成对话框的全部输入

输出操作,在用户按两个键之一时,将结束这一方法的执行,按链的值作为函数的值返回。

如果用户按 OK 键,调用 GetData 方法,将输入的方法存入。然后,用这些值调用显示卡片集的 Replace 方法。整个序列在一个循环中,这样,用户可输入多个卡片。

对于这些显示卡片的显示,要选择不同的方法。对问题有一个对话框,答案也的一个,这两个对话框必须一起工作。一种方法是为对话框定义一个全局变量,在对话框之外处理现场。下面是要求的全局变量:

Var

TheCard:PFlashCard;

Const

QBox:PDIALOG = nil;

ABos:PDIALOG = nil;

这两个对话框中的第一个是在显示卡片的提问一侧显示。它可以是这样的:

Procedure ShowQuestion;

Var

R : TRect;

Control : Word;

CardNumber : String;

Begin

(\* create the dialog box \*)

R.Assign(5, 2, 75, 11));

QBox := New(PDialog, Init(R, 'Question'));

With QBox ^ Do

Begin

(\* insert the question as static text \*)

R.Assign(4, 2, 63, 3);

```

Insert(New(PStaticText, Init(R, TheCard^.GetQuestion)));

(* include a Cancel button to allow the user to stop *)
R.Assign(40, 5, 50, 7);
Insert(New(PButton, Init(R, 'Cancel', cmCancel, bfNormal)));

(* include a Show Answer button for the ShowAnswer command *)
R.Assign 93, 5, 18, 7);
Insert(New(PButton, Init(R, 'Show ~A~nswer',
                        cmShowAnswer, bfdefault)));

(* insert the card number as static text *)
R.Assign(55, 5, 65, 6));
Str(TheCard^.GetNumber, CardNumber);
Insert(New(PStaticText, Init(R, 'Card: ' + CardNumber)));
End;

(* add the dialog box to the desktop (don't make it modal) *)
Desktop^.Insert(QBox);
End;

```

对话框本身没有新类型的项，它有两个静态文本和两个键。用户不能输入任何数据，不需要前面对话盒中的那么多的代码。在这个对话框中有一个新的命令常数：cmShowAnswer。对这个命令没有菜单或状态行项，但却是实现所要求的。在对话框平台上没有加上调用 ExecView，而是调用 Insert。把 Insert 方法加到对话框上后，允许在对话框外处理现场。这种方法不是标准的对话框形式，但是必须的，因为希望在显示答案时，问题盒不要删掉。

如果用户选择显示答案键，程序将在答案一侧显示。这由下面过程完成。

```

Procedure ShowAnswer;

Var
  R : TRect;
  Control : Word;

Begin

  (* create the dialog box *)
  R.Assign(5, 13, 75, 22);
  ABox := New(PDialog, Init(R, 'Answer'));
  With ABox^ Do
    Begin

```

```

    (* insert the answer as static text *)
    R.Assign(4, 2, 63, 3);
    Insert(New(PStaticText, Init(R, TheCard^.GetAnswer)));

    (* include a Next Question button *)
    R.Assign(25, 5, 42, 7);

    Insert(New(PButton, Init(R, '~N~ext Question',
        cmNextCard, bfNormal)));

    (* include a Cancel button *)
    R.Assign(47, 5, 58, 7);
    Insert(New(PButton, Init(R, 'Cancel', cmCancel, bfNormal)));

    (* include a Cancel button *)
    R.Assign(3, 5, 23, 7);
    Insert(New(PButton, Init(R, '~R~andom Question',
        cmRandomCard, bfDefault)))

    End;

    (* add the dialog box to the desktop *)
    DeskTop^.Insert(ABox);
    End;

```

对话框本身不提供使用何新的东西。逻辑由用户按键时出现的现场定义。在研究这一逻辑之前，先看最后这三个命令过程：

```

Procedure NextCard(Var TheSet : TFlashCardSet);

Begin
    TheCard := TheSet.NextCard;
    RemoveBoxes;
    ShowQuestion;
    End;

    ( * * * * * )

```

```

Procedure RandomCard(Var TheSet : TFlashCardSet);

```

```

Begin

```

```

TheCard := TheSet.AnyCard;
RemoveBoxes;
ShowQuestion;
End;

```

```

( * * * * * )

```

```

Porcedure RemoveBoxes;

```

```

Begin
Desktop ^ . Delete(ABox);
If ABox <> Nil Then
    Begin
        Dispose(ABox, Done);
        ABox := Nil;
    End;
Desktop ^ . Delete(QBox); If QBox <> Nil Then
    Begin
        Desktop ^ Delete(QBox); Distpse(ABox, Done);
        ABox := Nil;
    End;
End;

```

上面最后一个过程 RemoveBoxes 也不是主菜单的一部分，但是加上它是为了支持非标准对话框。

这些命令的工作方式如下：

1. 用户首先选择 Random Card 或 Next-Card。这两个选择将清平台并显示一个问题。
2. 用户若希望看到答案，按相应的键，产生 cmShowAnswer 现场，显示答案。
3. 若用户不想看答案，用户可从选择下列三项之一：
  - 按 Cancel 键，清除平台。
  - 按 F8 或 RandomCard 菜单命令，显示另一个问题。但不显示答案。
  - 按 F9 或 Next Card 菜单命令，显示下一个问题，但不显示答案。如果答案是明显的，用户可以选择后两种选择项。

1. 如果显示在答案一侧时，用户同样可以有这三种选择，这些选择项都作为按键包括在对话框中。

TButton 对象小结(DIALOGS.TPU)

继承谱系：

TButton→TView→TObject

主要方法：

Constructor Int(Var Bounds;TRect;Atitle:TTitlestr; Acommand;Word;AFlags;Byte);

建立加到对话框中的按钮对象。每个按钮都有名字和现场命令。

有关常数:

`bfNormal = $00;`

非缺省中心按钮标志。

`bfDefault = $01;`

缺省中以按钮标志。

`bfLeftJust = $02;`

非缺省左调整按钮标志。

`bfDefault + bfLeftJust`

缺省, 左调整键的结合

其它方法:

`Destructor Done; Virtual;`

释放按钮占用的域。

`Procedure Draw; Virtual;`

内部调用, 显示按钮

`Function GetPalette; PPalette; Virtual;`

可以覆盖, 改变按钮的色彩框架。

`Procedure HandleEvent (Var Event; TEvent); Virtual;`

内部调用, 确定是否按了键。

`Constructor Load (Var S; TStream);`

内部调用, 从流中装载按钮。

`Procedure MakeDefault (Enable; Boolean);`

调用这个过程, 使按钮为缺省, 或从缺省分配表中去掉。

`Procedure Setstate (Astate; Word; Enable; Boolean); Virtual;`

内部调用, 改变按钮的状态。

`procedure Store (Var S; TStream);`

内部调用, 把一个按钮送入流中。

`TInputLine` 对象小结(DIALOGS.TPU)

继承谱系:

`TInputLine → TView → TObject`

主要方法:

`Constructor Init (Var Bounds; TRect; AMaxLen; Integer);`

建立行输入对象, 最多允许输入 AMaxLen 个字符。

其它方法:

`Function DataSize; Word; Virtual;`

内部调用, 得到输入行的最大尺寸。建立处理其它数据类型的后代对象时, 可以覆盖这个函数。

`Destructor Done; Virtual;`

释放行输入对象。

`procedure Draw; Virtual;`

内部调用, 显示输入行。在建立处理其它数据类型的后代对象时, 可以覆盖这个过程。

Procedure GetData(Var Rec); Virtual;

可用于把数据类型转换为适于编辑的字符串。

Function Getpalette:PPalette;Virtual;

可用于改变输入行的色彩框架。

Procedure HandleEvent(Var Event:TEvent); Virtual;

内部调用, 识别输入行中的现场。

Constructor Load(Var s:TStream);

内部调用, 从流中装载一输入行。

Procedure SelectAll(Enable:Boolean);

使整个输入行为高亮度, 或使其为非高亮度。

Procedure SetData(Var Rec);Virtual;

可用于将输入的字符串转换为用户定义的数据类型。

Procedure Setstate(AState:Word;Enable:Boolean);Virtual;

内部调用, 改变输入行的状态。

Procedure Store (Var S:TStream);

内部调用, 把输入行送到流中。

TLabel 对象小结(DIALOGS.TPU)

继承谱系:

TLabel→TStaticText→TView→TObject

主要方法:

Constructor Init(Var Bounds:TRect;AText:String; ALink:Pview);

建立一个标号对象, 当选择 ALink 视区时, 将为高亮度。

其它方法:

Destructor Done; Virtual;

释放标号对象。

Procedure Draw;Virtual;

内部调用, 显示标号。

Function GetPalette:Ppalette;Virtual;

可用于改变标号的色彩框架。

Procedure HandleEvent(Var Event:TEvent); Virtual;

内部调用, 识别标号中的现场。

Constructor Load(Var S:TStream);

内部调用, 从流中装载标号。

Procedure Store(Var S:TStream);

内部调用, 将一个标号送入流中。

TStaticText 对象小结

继承谱系:

TStaticText→TView→TObject

主要方法:

Constructor Init(Var Bounds:TRect;AMaxLen:Integer);

建立一个不可选择,不能改变的静态文本行。

其它方法:

Destructor Done;Virtual;

释放静态文本对象。

Procedure Draw;Virtual;

内部调用,显示静态文本。

Function GePalette:PPalette;Virtual;

可用于改变输入行的色彩结构。

Procedure GetText(Var S:String); Virtual;

内部调用,得到存贮的字符串。

Constructor Load(Var S:TStream);

内部调用,从流中装载静态文本。

Procedure Store (Var S:Tstream);

内部调用,把静态文本送入流中。

TCheckBoxes 对象小结(DIALOGS.TPU)

继承谱系:

TCheckBoxes→Tcluster→TView→TObject

主要继承方法:

从 TCluster 中继承的有:

Constructor Init(Var Bounds:TRect;Astring:PSItem);

其它方法:

Procedure Draw;Virtual;

内部调用,显示检查盒。

Function Mark (Item:Integer);Boolean;Virtual;

内部调用,看项是否检测。

Procedure Press(Item:Integer);Virtual;

内部调用,联接一项。

TRadioButton 对象小结(DIALOGS.TPU)

继承谱系:

TRadioButton→TCluster→TView→TObject

主要继承方法:

从 TCluster 中继承的有:

Constructor Init(Var Bounds:TRect;AStrings:PSItem);

其它方法

Procedure Draw;Virtual;

内部调用,显示 Radio 按键。

Function Mark (Item:Integer);Boolean;Virtual;

内部调用,看第 n 项是否选择。

Procedure Moved To(Item:Integer); Virtual;



内部调用, 选择越过第 n 项。

Procedure Press(Item: Integer); Virtual;

当一个键按下时, 由内部调用。

Procedure SetData(Var Rec); Virtual;

内部调用, 把按键信息复制到内部域。

TCluster 对象小结 (DIALOGS.TPU)

继承谱系:

Tcluster → TView → TObject

主要方法:

Constructor Init(Var Bounds: TRect; Astrings: PSItem);

建立一个簇对象。簇是一个抽象对象, 用于定义检查盒和 radio 按键。

有关数据类型:

PSItem = ^ TSItem;

TSItem = ^ Record

Value: Pstring (\* pstring + ^ String \*)

Next: PSItem;

End;

定义簇对象的字符串链表。

有关函数:

Function NewSItem(Str: String; ANext: PSItem): PSItem;

分配内存, 并将内含字符串赋给 TSItem 记录。

其它方法:

Function DataSize: Word; Virtual;

内部调用, 得到簇中的项数。

Destructor Done; Virtual;

释放簇对象。

Procedure DrawBox; Virtual;

内部调用, 显示下一项的盒。

Procedure GetData(Var Rec); Virtual;

内部调用, 从内部域复制簇信息。

Function GethelpCtx: Word; Virtual;

允许基于选择簇项的不同的帮助相关文本。

Function GetPalette: Palette; Virtual;

可用于改变簇的色彩结构。

Procedure HandleEvent(Var Event: TEvent); Virtual;

内部调用, 识别簇中的现场。

Constructor Load(Var S: Tstream);

内部调用, 从流中装载簇。

Function Mark(Item: Integer): Boolean; Virtual;

内部调用, 看哪一个项目作了标记。

Procedure MovedTo(Item:Integer);virtual;

内部调用, 选择越过第 n 项。

Procedure Press(Item:Integer);Virtual;

内部调用选择第 n 项。

Procedure SetData(Var Rec); virtual;

内部调用, 把簇信息复制到内部结构中。

Procedure setstate(Astate:Word; Endble:Boolean);Virtual;

内部调用, 改变簇的状态。

procedure Store(Var S:TStream);

内部调用, 将一个簇送到流中。

TListBox 对象小结(DIALOGS.TPU)

继承谱系:

TListBox→TListViewer→TView→TObject

主要方法:

Constructor Init(Var Bounds:TRect;ANumCols:Word; VscrollBar:PScrollBar);

以给定大小和列数建立一个列表盒。可有一个选择垂直滚动条。

Procedure NewList(Alist;PCollection); Virtual;

将集合中的项赋值给列表盒。

其它方法:

Function DataSize:Word; Virtual;

内部调用, 得到列表盒的当前尺寸。

Destructor Done;Virtual;

释放列表盒对象。

Procedure GetData(Var Rec); Virtual;

内部调用, 从内部结构中复制列表项。

Function GetTEXT(Item:Integer;MaxLen:Integer): String; Virtual;

内部调用, 得到表中的第 n 项。

Constructor Load(Var S:TStream);

内部调用, 从流中装载一个列表盒。

procedure SetData(Var Rec);Virtual;

内部调用, 把列表项复制到内部结构中。

Procedure Store(Var S:TStream);

内部调用, 将一个列表盒送到流中。

THistory 对象小结(DIALOGS.TPU)

继承谱系。

THistory→TView→TObject

主要方法:

Constructor Init(Var Bounds:TRect;ALink:PInputLine; AHistoryID:Word);

建立与给定输入行相关的采集列表。参数 AHistoryID 允许多个输入行占用同一个历史列表。这个表为一个单一的图像(下箭头), 直到选择这个图像。

其它方法:

Procedure Draw; Virtual;

内部调用, 显示历史图像。

Function GetPalette: PPalette; Virtual;

可用于改变历史图像的色彩框架。

Constructor Load (Var S: TStream);

内部调用, 从流中装载历史列表。

procedure Store (Var S: TStream);

内部调用, 把历史列表送到流中。

TFileDialog 对象小结 (STDDLG, TPU)

继承谱系:

TFileDialog → TDialog → TWindow → TGroup → TView → TObject

主要方法:

Constructor Init (AWildCard: WildStr; ATitle: String; InputName: String; Buttons: Word; HistoryId: Byte);

建立带文件列表盒, 历史列表及一个或多个标准按键的对话框。

procedure GetFileName (Var S: PathStr);

返回用户选定的文件名。

相关类型:

WildStr = Dos.PathStr;

定义文件名的初始值, 通常含有通配符。

相关常数:

FdOKButton = 1;

包含 OK 按键的标记。

fdOpenButton = 2;

包含 Open 按键的标记。

fdReplaceButton = 4;

包含 Replace 按键的标记。

fdClearButton = 8;

包含 Clear 按键的标记。

其它方法:

Destructor Done; Virtual;

释放文件对话框。

Procedure GetData (Var Rec); Virtual;

内部调用, 从内部结构中恢复数据。

Procedure HandleEvent (Var Event: TEvent); Virtual;

内部调用, 处理文件对话框中的现场。

Constructor Load (Var S: TStream);

内部调用, 从流中读出文件对话框。

Procedure Store (Var S: TStream);

内部调用，将文件对话框数据写到流中。

```
Function Valid(command:Word):Boolean;Virtual;
```

内部调用，确定哪个命令有效。

其它的主要命令有把一个卡片存到一个文件中再装入它们。Turbo Vision 为此提供了一个极为强有力的对象，称作流。流是一个路径，对象沿着它送到另一个地方，或从另一个地方取得。支持流的主要有 DOS 文件或扩展内存。流象集合一样是多态的，允许对不同的对象类型使用同一个流。

显示卡片应用程序使用了缓冲 DOS 文件流。这个对象类型称作 TBufStream。为了在用户建立的对象中使用这个流，必须把这些对象登记在 Turbo Vision 中。登记的第一步是定义一个登记记录。下面是显示卡片的记录：

```
Const
(* 用于流的登记 *)
RFlashCard: TStreamRec = C
    Objtype: 1050;
    VMtLink: OfS(KindOf(TFlashCard)^);
    Load: @TFlashCard.Load;
    Store: @TFlashCard.Store);
```

登记记录的第一个域是唯一一个对象的 ID，其有效值为从 1000 到 65535。第二个域指向对象的 VMT。最后两个域是这个对象类型访问流的过程。对于这种对象类型的 Load 和 Store 的实现如下：

```
Constructor TFlashCard.Load(Var S : TStream);

Begin
S.Read(Number, SizeOf(Number));
S.Read(Question, SizeOf(Question));
S.Read(Answer, SizeOf(Answer));
End;

( * * * * * )

Procedure TFlashCard.Store(Var S : TStream);

Begin
S.Write(Number, SizeOf(Number));
S.Write(Question, SizeOf(Question));
S.Write(Answer, SizeOf(Answer));
End;
```

这两个过程的实现主要是对对象类型的每个数据域调用 TStream.Read 和 TStream.Write。如果对象是另外对象的后代，这些覆盖方法必须调用其祖先的对应过程。TFlashCard-set 对象类型也需要登记和支持过程：

```

Const
(* for Stream registration *)
  RFlash/cardSet; TStreamRec = (
    ObjType: 140;
    VmtLink: ofs(KindOf(TFlashCardSet) ^ );
    Lpad: @TFlashCardSet.Load;
    Store: @TFlashCardSet.Store);

( * * * * * )

Constructor TFlashCardSet.Load(Var S : TStream);

Begin
  TCollection.Load(S);
  CurrentCard := -1;
  Randomize;
End;

( * * * * * )

Procedure TFlashCardSet.Store(Var S : TStream);

Procedure TFlashCardSet.Store(Var S : TStream);

Begin
  TCollection.Store(S);
End;

```

只要在应用程序的结构中设置了对象，就必须登记流，通常在应用程序的开头由一个过程进行。流中用到的所有对象类型都应登记。在显示卡片应用程序中，需要登记的类型有 TFlashCard, TFlashCardset 和 TCollection:

```

Procedure RegisterStream;
Begin
  RegisterType(Rcollection);
  RegisterType(RFlashCard);
  RegisterType(RFlashCardset);
End;

```

在显示卡片程序中，流用于 Save File 和 Open File 命令。只有用应用程序存贮的文件可以读到应用程序中。

Turbo Vision 提供了一个完整的对话框,用于在用户程序中用的 IDE。TFileDialog 对象类型是前面的总结。用这个对话框写过程非常方便:

```
Procedure SaveFile(Var TheSet : TFlashCardSet);

Var
  SaveBox : PFileDialog;
  FCStream : TBufStream;
  Control : Word;
Begin

  (* use the standard Save File dialog box *)
  New(SaveBox, Init(TheFileName, 'Save File As',
    'Save File Name', fdOkButton, 1));
  Control := Desktop^.ExecView(SaveBox);
  If Control <> cmCancel Then
    Begin

      (* save the sset using a buffered stream *)
      SaveBox^.GetFileName(TheFileName);
      FCStream>Init(TheFileName, stCtreate, 512);
      FCStrea.Put(@TheSet);
      FCStrea.Done;
    End;
  End;
```

SaveFile 过程首先建立对话框,要求的初始化参数是缺省文件名,对话框标题,文件名标号,所用按键的列表和历史列表 ID。然后执行对话框,并返回由用户按键产生的现场。

假设用户想把卡片集存入流中,过程得到文件名,把文件作为流打开,用显示卡片集的地址调用 TBufStream.Put。方法 Put 调用 TFlashCardSet 对象类型的 Store 过程,即对每个显示卡片顺序调用 Store 过程。

从流中装载数据是一个非常类型的过程。首先对话框得到一个文件名。第二步,用文件名建立一个流。最后,从流中装载数据。下面是打开显示卡片文件的代码:

```
Procedure OpenFile(Var NewSet : TFlashCardSet);

Var
  OpenBox : PfileDialog;
  FCStrea : TBufStream;
  Control : Word;
```

```
PFCSet : PFlashCardSet;
```

```
Begin
```

```
( * use the standard Open File dialog box * )
```

```
New(OpenBox, Init('*.CRD', 'Open File', Card File Name',  
    fdOkButton + fdOpenButton, 1));
```

```
Control := DeskTop^.ExecView (OpenBox);
```

```
If Control <> cmCancel Then
```

```
Begin
```

```
( * read the file as a buffered stream * )
```

```
OpenBox^.GetFileName(TheFileName);
```

```
FCStream.Init(TheFileName, ae, stOpenRead, 512);
```

```
NewSet := PFCSet^;
```

```
End;
```

```
End;
```

完成全部显示卡片应用程序工作的最后一步是修改主应用方法，支持实际命令。首先需要一些变量：

```
Var
```

```
FlashCardAp: TFlashCardApp;
```

```
FlashCards: TFlashCardSet;
```

```
OneCard: TFlashCard;
```

接下来，必须加上现场命令所需的参数并加上对这些新命令的支持：

```
Procedure TFlashCardApp.HandleEvent(Var Event: TEvent);
```

```
Begin
```

```
if Application.HandleEvent(Event);
```

```
If Event.What = evCommand Then
```

```
Begin
```

```
Case Event.Command Of
```

```
cmFileOpne : OpenFile(FlashCards);
```

```
cmFileSave : SaveFile(FlashCards);
```

```
cmEditAdd : AddCard(FlashCards);
```

```
cmEditDelete : ChangeCard;
```

```
cmNextCard : NextCard(FlashCards);
```

```
cmRandomCard : RandomCard(FlashCards);
```

```
cmShowAnswer : ShowAnswer;
```

```
cmCancel : RemoveBoxes;
```

```

    Else
        Exit;
    End;
    ClearEvent(Event);
End;

End;

```

最后，加上主程序初始化所需的代码。最后的程序代码如下：

```

Begin
FlashCardS. Init;
RegistorStreams;
FlashCardApp. Init;
FlashCardApp. Run;
FlashcARApp. Done;
End;

```

现在显示卡片应用程序已经完成。但是为了简单，有许多东西，如错误检查，已经去掉。

**TBufStream 对象小结(OBJECTS.TPU)**

**继承谱系：**

TBufSteam→TDosStream→TStream→TObject

**主要方法：**

Constructor Init(FileName: FNameStr; Mode, Size :Word);

用给定的文件模式和缓冲区大小打开或建立一个作为流的文件。

Destructor Done;Virtual;

关闭文件并刷新缓冲区。

**主要继承方法：**

从 TStream 中继承的有：

Function Get:PObject;

Procedure Put(P:PObject);

**有关常数：**

StCreate= \$ 3C00;

建立或重写文件的模式。

StOpenRead= \$ 3D00;

作为只读打开文件的模式。

StOpenWrite= \$ 3D01;

作为读

写打开文件的模式。

**相关类型：**

FNamestr=String[79];

文件名最多可有79 个字符。



其它方法:

Procedure Flush: Virtual;

内部调用, 刷新文件缓冲区。

Function GetPos: Longint; Virtual;

内部调用, 确定文件的位置。

Function GetSize: Longint; Virtual;

内部调用, 确定文件的大小。

Procedure Read (Var Buf; Count: Word); Virtual;

内部调用, 到文件的位置。

Procedure Truncate; Virtual;

内部调用, 删除到文件尾。

Procedure Write (Var Buf; Count: Word); Virtual;

内部调用, 向一个文件中写字节。

TEmsStream 对象小结 (OBJECT.TPU)

继承谱系:

TEmsStream → TStream → TObject

主要方法:

Constructor Init (MinSize: Longint);

用给定字节数建立 EMS 流。

Destructor Done; Virtual;

释放 EMS 内存时必须调用。

主要继承方法:

从 TStream 中继承的有:

Function Get: PObject;

Procedure Put (P: PObject);

其它方法:

Function GetPos: Longint; Virtual;

内部调用, 确定当前 EMS 地址。

Function GetSize: Longint; Virtual;

内部调用, 确定 EMS 流的大小。

Procedure Read (Var Buf; Count: Word); Virtual;

内部调用, 到 EMS 内存中一个特定地址。

Procedure Truncate; Virtual;

内部调用, 去掉当前 EMS 地址之后的全部数据。

Procedure Write (Var Buf; Count: Word); Virtual;

内部调用, 向 EMS 内存写字节。

TStream 对象小结 (OBJECT.TPU)

继承谱系:

Tstream → TObject

主要方法:

Function Get;PObject;

通过对对象的每一部分调用构造 Load, 从流中读一个对象。

procedure Put(P:PObject);

通过对对象的每一部分调用过程 Store, 把一个对象写到流中

Procedure Read(Var Buf;Count;Word); Virtual;

从流中读给定大小的变量, 由每个 Load 过程调用。

procedure Write(Var Buf; Count;Word);Virtual;

向流中写给定大小的变量, 由每个 Store 过程调用。

相关过程:

procedure Register Type(Var S: TStreamRec);

允许由 TObject 中继承的对象类型用流装载或存贮。

相关类型:

PstreamRec = ^ TStreamRec;

TstreamRec = Record

ObjType: word;

VmtLink: Word;

Load: Pointer;

Store: PPointer

Next: Word;

End;

定义对象的流特性。

其它方法:

流对象几乎都未确切建立, 所以其内部方法没有多大价值。这些方法可见每个后代对象类型。

除了上面介绍的对象外, Turbo Vision 还有其它一些对象, 但是由于在上面的应用程序中未用到, 所以还有介绍。下面介绍其中的四个对象, 它们对开发应用程序是有用的。

如果显示大量文本, 或允许用户输入大量文本时, 应建立 Twindow 的后代对象。如果窗口中的信息不能一次显示在屏幕上, 必须把 Tscroller 型视像对象加到窗口中。物理滚动光条是 TSCrollerBar 型。源是带缓冲 DOS 流的个特例, 用于保存或恢复用户接口对象, 如菜单和对话框, 可以把建立这些对象的代码写到一个源文件中, 主应用程序只要简单地从源文件中读出这些对象。这可以使程序更模块化, 也可以减小主执行程序的大小。

Twindow 对象小节 (VIEWS.TPU)

继承谱系:

TWindow → TGroup → Tview → TObject

主要方法:

Costructor Init(Var Bound; TRect; Atitle; TTitlestr; ANumber; Integer);

用给定标题和窗口号在指定区域建立窗口。

Function StandardScrollBar(AObjects: Word); PScrollBar;

在窗口中建立并插入一个滚动光条。

相关常数:

**WfMove=1;**  
 允许窗口在平台上移动。  
**WfGrow=2,**  
 允许窗口改变大小。  
**WfClose=4;**  
 在窗口的右上角加的一个封闭图像。  
**SfZoom=8;**  
 在窗口的有右上角加一个移动图像。  
 其它方法:  
**Procedure Close;Virtual;**  
 内部调用, 关闭窗口。  
**Destructor Done;Virtual;**  
 释放窗口对象。  
**Function GetPalette;PPalette;Virtual;**  
 可以覆盖, 改变窗口的颜色。  
**Function GetTitle(MaxSize;Integer); TtitleStr, Virtual;**  
 内部调用, 得到窗口标题。  
**procedure Handle Event(Var Event;TEvent); Virtual;**  
 内部调用, 处理窗口中出现的现场, 如移动, 改变大小等。  
**Procedure InitFrame; Virtual;**  
 内部调用, 建立窗口框架。  
**Constructor Load(Var S;TStream);**  
 内部调用, 从流中读窗口。  
**Procedure SetState(Astate;Word;Enable;Boolean);Virtual,**  
 内部调用, 改变窗口状态。  
**Procedure SizeLimits(Var Min, Max;TPoint); virtual;**  
 改变窗口大小界限。  
**Procedure Store(**  
**Var S;TStream);**  
 内部调用, 把窗口写到流中。  
**procedure Zoom;Virtual;**  
 内部调用, 放大窗口。  
**TScroller 对象小结(VIEWS.TPU)**  
 继承谱系:  
**TScroller→TView→TObject**  
 主要方法:  
**constructor Init(Var Bounds;TRect;AHScrollBar, AvScrollBar;PScrollBar),**  
 用选择滚动光条在窗口中建立滚动器。  
**Procedure SetLimit(X,Y;Integer);**  
 设置滚动区中数据的大小。注意不是滚动区的大小。

主要继承方法:

从 TView 中继承的有:

procedure Draw; Virtual;

其它方法:

Procedure ChangeBounds(Var Bounds:TRect); Virtual;

内部调用, 改变滚动器的大小。

Function GetPalette; PPalette; Virtual;

可以覆盖, 改变滚动器的颜色。

Procedure Handle—Event(Var Event:TEvent); Virtual;

内部调用, 处理滚动器现场。

Constructor Load(Var s:TStream);

内部调用, 从流中读一个滚动器对象。

procedure ScrollDraw; Virtual;

内部调用, 恢复滚动器

procedure Scrollto(X,Y:Integer);

内部调用, 滚动到视区的不同部分。

procedure Setstate(Astate;word; Enable:Boolean);

内部调用, 改变滚动器的状态。

Procedure Store(Var S:Tstream);

内部调用, 把滚动器写到流中。

TScrollBar 对象小结 (VIEWS.TPU)

继承谱系:

TScrollBar→Tview→TObject

相关常数:

SbHorizontal=0;

允许有水平滚动光条。

SbVertical=1;

允许有垂直滚动光条。

SbHandleKeyboard=2;

允许支持滚动键盘命令。

其它方法:

TScrollBar 型对象几乎都是用 TWindow.StandardScrollBar 方法建立的。

TResourceFile 对象小结 (OBJECTS.TPU)

继承谱系:

TResource—File →TObject

主要方法:

Constructor Init(Astream;Pstream);

用带缓冲 DOS 流初始化源文件。

Function Get (Key:String): PObject;

从源文件中恢复对象。

**Procedure Put (Item : PObject ; Key : String) ;**

将一个对象写到源文件中，并赋给这个对象一个关键字。

其它方法：

**Function Count ; Integer ;**

确定源文件中对象的数量。

**Procedure Delete (Key : String) ;**

删除源文件中储存的一个对象。

**Destructor Done ; Virtual ;**

释放源文件，刷新缓冲区。

**Procedure Flush ;**

如果作了任何改变，刷新源文件缓冲区。

**Function KeyAt (I : Integer) : String ;**

可用于确定源文件的内容。

**Function SwitchTo (Astream : Pstream ; Pack : Boolean) :**

**PStream ;**

把源文件从一个流转向另一个，如果需要的话，存储这个新文件。

录

第一章 Turbo Pascal 编程的基本概念

- § 1.1 Turbo Pascal 程序的一般形式
- § 1.2 Turbo Pascal 与标准 Pascal
- § 1.3 程序结构
  - § 1.3.1 程序头与编译指令
  - § 1.3.2 数据部分
  - § 1.3.3 代码部分
  - § 1.3.4 包含文件
  - § 1.3.5 覆盖块
  - § 1.3.6 过程与函数

第二章数据类型与表达式

- § 2.1 常量标准数据类型
- § 2.2 常量
- § 2.3 用户定义的数据类型
- § 2.4 集合
- § 2.5 数组
- § 2.6 记录
- § 2.7 turbo Pascal 中的表达式
  - § 2.7.1 算术运算
  - § 2.7.2 整型运算
  - § 2.7.3 算术函数
  - § 2.7.4 逻辑运算
  - § 2.7.5 集合运算
- § 2.8 类型间的关系

第三章程序的控制结构

- § 3.1 程序的选择结构
- § 3.2 程序的循环结构
  - § 3.2.1 For—Do 循环
  - § 3.2.2 Repeat—Until 循环
  - § 3.2.3 While - Do 循环
- § 3.3 非结构分枝

第四章 Turbo Pascal 的集成开发环境

- § 4.1 File 菜单
- § 4.2 Edit 菜单
- § 4.3 Search 菜单
  - § 4.3.1 Find...CTRL - QE
  - § 4.3.2 Replace...CTRL - QA
  - § 4.3.3 Search again CTRL - L
  - § 4.3.4 Go to line number
  - § 4.3.5 Find Procedure
  - § 4.3.6 Find error...ALT...F8
- § 4.4 Run 菜单
- § 4.5 Compile 菜单
- § 4.6 Debug 菜单
- § 4.7 Options 菜单
  - § 4.7.1 Compiler
  - § 4.7.2 Memory Size
  - § 4.7.3 LinrRer
  - § 4.7.4 Directories
  - § 4.7.5 Environment
  - § 4.7.6 Save Options
  - § 4.7.7 Retrieve Options
- § 4.8 window 菜单

第五章指针与动态内存分配

- § 5.1 Turbo Pascal 的内存分配

§ 5 . 2 堆和指针	
§ 5 . 3 链表	
§ 5 . 4 树	
§ 5 . 5 操作符	
第六章文件	
§ 6 . 1 文本文件	
§ 6 . 2 类型文件	
§ 6 . 3 无类型文件	
§ 6 . 4 缓冲区	
§ 6 . 5 文件的删除与改名	
第七章外部过程、过程与函数库	
§ 7 . 1 嵌入代码	
§ 7 . 2 外部过程	
§ 7 . 3 Turbo Debugger	
§ 7 . 4 视频显示基本例程	
§ 7 . 5 带缓冲字符串输入	
§ 7 . 6 大字符串的处理	
§ 7 . 7 算术函数	
§ 7 . 8 文件加密	
第八章 Turbo Pascal 工具箱	
§ 8 . 1 数据库工具箱	
§ 8 . 2 图形工具箱	
§ 8 . 3 编辑工具箱	
§ 8 . 4 数值方法工具箱	
第九章编程技术	
§ 9 . 1 字符串	
§ 9 . 2 递归	
§ 9 . 3 DOS 设备	
§ 9 . 4 合并	
§ 9 . 5 排序	
§ 9 . 6 搜索	
第十章视频：文本显示与图形	
§ 10 . 1 视频显示的基本概念	
§ 10 . 2 使用 Turbo Pascal 显示文本	
§ 10 . 2 . 1 方式、颜色和位置控制	
§ 10 . 2 . 2 直接存取视频存贮区	
§ 10 . 2 . 3 Turbo Pascal 的窗口程序设计	
§ 10 . 2 . 3 . 1 弹出窗口	
§ 10 . 2 . 3 . 2 多逻辑屏幕和弹出窗口	
§ 10 . 3 Turbo Pascal 的图形单元 (GRAPH)	
§ 10 . 3 . 1 描点	
§ 10 . 3 . 2 画线	
§ 10 . 3 . 圆、直线和图形模式的综合使用	
§ 10 . 3 . 4 图形文本	
§ 10 . 3 . 5 多边形及填彩	
第十一章DOS：软中断与硬中断	
§ 11 . 1 DOS与BIOS服务	
§ 11 . 2 DOS与中断	
§ 11 . 3 软中断：DOS单元与操作系统公共服务	
§ 11 . 3 . 1 Turbo Pascal 的DOS单元	
§ 11 . 3 . 1 DOS单元的常量与类型	
§ 11 . 3 . 1 . 2 DOS单元的DOSError 变量	
§ 11 . 3 . 1 . 3 DOS单元的过程与函数	
§ 11 . 3 . 2 直接访问BIOS和DOS 服务	
§ 11 . 3 . 2 . 1 磁盘驱动器服务	
§ 11 . 3 . 2 . 2 视频服务	
§ 11 . 3 . 2 . 3 时间和日期功能	
§ 11 . 3 . 2 . 4 报告换标状态	
§ 11 . 3 . 3 使用DOS单元中其它例程	
§ 11 . 4 硬中断，远程通信与TSR实现	

- § 1 1 . 4 . 1 编写中断处理程序
- § 1 1 . 4 . 2 P C 远程通信及程序
- § 1 1 . 4 . 3 内存驻留程序 ( T S R ) 实现
- § 1 1 . 4 . 3 . 1 解决再入问题
- § 1 1 . 4 . 3 . 2 用 T u r b o P a s c a l 实现 T S R

## 第十二章 优化与调试

- § 1 2 . 1 控制结构的优化
- § 1 2 . 2 其它优化方法
- § 1 2 . 3 编译指令
- § 1 2 . 4 调用与参数
- § 1 2 . 5 T u r b o P a s c a l 调试器

## 第十三章 对象

- § 1 3 . 1 对象的概念
- § 1 3 . 2 继承
- § 1 3 . 3 封装
- § 1 3 . 4 静态方法和虚拟方法
- § 1 3 . 5 对象类型的兼容性
- § 1 3 . 6 对象的动态分配
- § 1 3 . 7 多态性

## 第十四章 T u r b o V i s i o n